Integer and polynomial multiplication

Romain Lebreton
–
Équipe ECO

November 18th, 2021

Context of today's talk

In 1 second, we can multiply integers of 30 000 000 *digits*
and polynomials of degree 500 000.

- Mini-course (L2, M1, M2 & bonus)
- Theoretical and practical aspects
- Presentation of team research topics
  - Link with current research
  - Relation with implementations (LinBox)

Application / Motivation

- Exact computation:
  - Many operations reduced to multiplication:
    exponentiation, division, pgcd, factorization, …
  - Mathematical software: GMP, Sage, Matlab, Maple, …
- Other domains using exact computation:
  - Cryptography:
    [Discrete logarithm computation in a 180-digit prime field, 2014]
  - Combinatorics, Number Theory, Error Correcting Codes, …
- Numerical computation:
  - Trillions of digits of π [Yee, Kondo '11]
  - Robotics: Equilibrium of cable driven parallel robots

From polynomial to integer multiplication

To multiply

$$(794x^2 + 983x + 523) \quad \times \quad (564x^2 + 637x + 185)$$

Evaluate at $x = 10^7$

$$\begin{array}{rcl} (794 \cdot (10^7)^2 + 983 \cdot 10^7 + 523) & \times & (564 \cdot (10^7)^2 + 637 \cdot 10^7 + 185) \\ 794000009830000523 & \times & 564000006370000185 \\ & = & \end{array}$$

$$447816106019010680330515006009675 5$$

"Interpolate" at $x = 10^7$

$$447816x^4 + 1060190x^3 + 1068033x^2 + 515006x + 96755$$

Remarks:

- Technique called Kronecker substitution (1882)
- 10^7 is the minimal power of 10 greater than all coefficients

From integer to polynomial multiplication

To multiply

$$794983523 \quad \times \quad 564637185$$

"Interpolation" at $x = 10^3$

$$(794x^2 + 983x + 523) \quad \times \quad (564x^2 + 637x + 185)$$
$$=$$
$$447816x^4 + 1060190x^3 + 1068033x^2 + 515006x + 96755$$

Evaluate at $x = 10^3$

$$447816 \cdot 10^{12} + 1060190 \cdot 10^9 + 1068033 \cdot 10^6 + 515006 \cdot 10^3 + 96755$$
$$=$$
$$448877258548102755$$

From integer to polynomial multiplication

Compute

$$447816\cdot(10^3)^4+1060190\cdot(10^3)^3+1068033\cdot(10^3)^2+515006\cdot10^3+96755$$

$$\begin{array}{cr}
 & 96\,755 \\
+ & 515\,006\,000 \\
+ & 1\,068\,033\,000\,000 \\
+ & 1\,060\,190\,000\,000\,000 \\
+ & 447\,816\,000\,000\,000\,000 \\
\hline
= & 448\,877\,258\,548\,102\,755
\end{array}$$

Addition with carry:

- *Input:* $a, b \in [\![0, 999]\!]$ and a carry $c \in [\![0, 1]\!]$
  Output: $d \in [\![0, 999]\!]$ and a carry $e \in [\![0, 1]\!]$ s.t. $a + b + c = d + 1000e$.
- Base 2^{64} instead of base 1000

Outline

Polynomial multiplication algorithms:

1. Karatsuba
2. Fast Fourier Tranform (FFT)
3. Truncated FFT (TFT)

Polynomial data structures

- Dense polynomials = list of all coefficients

$$x^{11} + 5x^{10} + 9x^8 + 4x^7 + 7x^6 + x^2 + 8$$

1	5	0	9	4	7	0	0	0	1	0	8

Polynomial data structures

- Dense polynomials = list of all coefficients
- Sparse polynomials = list of all non-zero coefficients

$$x^{29} + 9x^{12} + 4x^{11} + 2x^2$$

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	4	0	0	0	0	0	0	0	0	2	0	0

29	12	11	2
1	9	4	2

Polynomial data structures

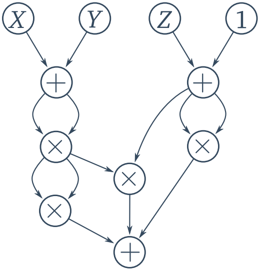
- Dense polynomials = list of all coefficients
- Sparse polynomials = list of all non-zero coefficients
- Straight-line programs

$$X^4+4X^3Y+6X^2Y^2+4XY^3+Y^4+X^2Z+2XYZ+Y^2Z+X^2+2XY+Y^2+Z^2+2Z+1$$

Polynomial data structures

- Dense polynomials = list of all coefficients
- Sparse polynomials = list of all non-zero coefficients
- Straight-line programs

$$((X+Y)^2)^2 + (X+Y)^2 \cdot (Z+1) + (Z+1)^2$$



Polynomial data structures

- **Dense polynomials = list of all coefficients**
- Sparse polynomials = list of all non-zero coefficients
- Straight-line programs

Naive polynomial multiplication

Example:

$$
\begin{array}{c} a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\ \times \\ b_0 + b_1 x + b_2 x^2 + b_3 x^3 \end{array}
=
\begin{array}{cccccccccc}
 & (& & & & & & & a_3 b_3 &) & x^6 \\
+ & (& & & & & a_2 b_3 & + & a_3 b_2 &) & x^5 \\
+ & (& & & a_1 b_3 & + & a_2 b_2 & + & a_3 b_1 &) & x^4 \\
+ & (& a_0 b_3 & + & a_1 b_2 & + & a_2 b_1 & + & a_3 b_0 &) & x^3 \\
+ & (& a_0 b_2 & + & a_1 b_1 & + & a_2 b_0 & & &) & x^2 \\
+ & (& a_0 b_1 & + & a_1 b_0 & & & & &) & x^1 \\
+ & (& a_0 b_0 & & & & & & &) & x^0
\end{array}
$$

Naive polynomial multiplication

Example:

$$\begin{array}{c} a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\ \times \\ b_0 + b_1 x + b_2 x^2 + b_3 x^3 \end{array} = \begin{array}{cccccccccc}
 & (& & & & & & & a_3 b_3 &) \; x^6 \\
+ & (& & & & & a_2 b_3 & + & a_3 b_2 &) \; x^5 \\
+ & (& & & a_1 b_3 & + & a_2 b_2 & + & a_3 b_1 &) \; x^4 \\
+ & (& a_0 b_3 & + & a_1 b_2 & + & a_2 b_1 & + & a_3 b_0 &) \; x^3 \\
+ & (& a_0 b_2 & + & a_1 b_1 & + & a_2 b_0 & & &) \; x^2 \\
+ & (& a_0 b_1 & + & a_1 b_0 & & & & &) \; x^1 \\
+ & (& a_0 b_0 & & & & & & &) \; x^0
\end{array}$$

In general: Multiplication of degree n polynomials in $O(n^2)$ arithmetic operations $(+, -, \times)$

Naive polynomial multiplication

Example:

$$\begin{array}{c} a_0 + a_1x + a_2x^2 + a_3x^3 \\ \times \\ b_0 + b_1x + b_2x^2 + b_3x^3 \end{array} = \begin{array}{clcccccccl} & (& & & & & & & a_3b_3 &)\ x^6 \\ + & (& & & & & a_2b_3 & + & a_3b_2 &)\ x^5 \\ + & (& & & a_1b_3 & + & a_2b_2 & + & a_3b_1 &)\ x^4 \\ + & (& a_0b_3 & + & a_1b_2 & + & a_2b_1 & + & a_3b_0 &)\ x^3 \\ + & (& a_0b_2 & + & a_1b_1 & + & a_2b_0 & & &)\ x^2 \\ + & (& a_0b_1 & + & a_1b_0 & & & & &)\ x^1 \\ + & (& a_0b_0 & & & & & & &)\ x^0 \end{array}$$

In general: Multiplication of degree n polynomials in $O(n^2)$ arithmetic operations $(+, -, \times)$

Lower bound: The multiplication costs at least n arith. operations

Naive polynomial multiplication

Example:

$$
\begin{array}{c}
a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\
\times \\
b_0 + b_1 x + b_2 x^2 + b_3 x^3
\end{array}
=
\begin{array}{cccccccccc}
 & (& & & & & & & a_3 b_3 &) \; x^6 \\
+ & (& & & & & a_2 b_3 & + & a_3 b_2 &) \; x^5 \\
+ & (& & & a_1 b_3 & + & a_2 b_2 & + & a_3 b_1 &) \; x^4 \\
+ & (& a_0 b_3 & + & a_1 b_2 & + & a_2 b_1 & + & a_3 b_0 &) \; x^3 \\
+ & (& a_0 b_2 & + & a_1 b_1 & + & a_2 b_0 & & &) \; x^2 \\
+ & (& a_0 b_1 & + & a_1 b_0 & & & & &) \; x^1 \\
+ & (& a_0 b_0 & & & & & & &) \; x^0
\end{array}
$$

In general: Multiplication of degree n polynomials in $O(n^2)$ arithmetic operations $(+, -, \times)$

Lower bound: The multiplication costs at least $n \log n$ arith. operations

Naive polynomial multiplication

Example:

$$\begin{array}{c} a_0 + a_1 x + a_2 x^2 + a_3 x^3 \\ \times \\ b_0 + b_1 x + b_2 x^2 + b_3 x^3 \end{array} = \begin{array}{cccccccccc} & (& & & & & & a_3 b_3 &) & x^6 \\ + & (& & & & & a_2 b_3 & + \; a_3 b_2 &) & x^5 \\ + & (& & & a_1 b_3 & + & a_2 b_2 & + \; a_3 b_1 &) & x^4 \\ + & (& a_0 b_3 & + & a_1 b_2 & + & a_2 b_1 & + \; a_3 b_0 &) & x^3 \\ + & (& a_0 b_2 & + & a_1 b_1 & + & a_2 b_0 & &) & x^2 \\ + & (& a_0 b_1 & + & a_1 b_0 & & & &) & x^1 \\ + & (& a_0 b_0 & & & & & &) & x^0 \end{array}$$

In general: Multiplication of degree n polynomials in $O(n^2)$ arithmetic operations $(+, -, \times)$

Lower bound: The multiplication costs at least $n \log n$ arith. operations

What is the best complexity of multiplication ?

Time estimates

Context: Trillions of digits of π [Yee, Kondo '11]

Time estimation:

- Computation equivalent to multiplying integers of 10^{12} digits
- *PC speed:* $\simeq 1$ GHz, so $\simeq 10^9$ op/s
-
  - Naive algorithm: $O(n^2)$
    $(10^{12})^2$ op, so $\frac{(10^{12})^2}{10^9}$ s $= 10^{15}$ s $\simeq 31\,000\,000$ years
  - Lower bound: $O(n \log n)$
    $12 \cdot 10^{12}$ op, so $\frac{12 \cdot 10^{12}}{10^9}$ s $= 12000$ s $\simeq 3.3$ hours

Outline

Polynomial multiplication algorithms:

1. **Karatsuba**
2. Fast Fourier Tranform (FFT)
3. Truncated FFT (TFT)

$$(a_0 + a_1 x) \cdot (b_0 + b_1 x) = c_0 + c_1 x + c_2 x^2$$

Naive algorithm: 4 multiplications

$$\begin{cases} c_0 &= a_0 b_0 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ c_2 &= a_1 b_1 \end{cases}$$

Karatsuba: 3 multiplications by writing

$$\begin{cases} c_0 &= a_0 b_0 \\ c_1 &= (a_0 + a_1) \cdot (b_0 + b_1) - a_0 b_0 - a_1 b_1 \\ c_2 &= a_1 b_1 \end{cases}$$

Is it better? Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Polynomials with 4 coefficients:

$$c_0 + \cdots + c_6x^6 := (\underbrace{a_0 + a_1x}_{a_l(x)} + \underbrace{a_2x^2 + a_3x^3}_{x^2a_h(x)}) \cdot (\underbrace{b_0 + b_1x}_{b_l(x)} + \underbrace{b_2x^2 + b_3x^3}_{x^2b_h(x)})$$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Polynomials with 4 coefficients:

$$c_0 + \cdots + c_6x^6 := (\underbrace{a_0 + a_1x}_{a_l(x)} + \underbrace{a_2x^2 + a_3x^3}_{x^2a_h(x)}) \cdot (\underbrace{b_0 + b_1x}_{b_l(x)} + \underbrace{b_2x^2 + b_3x^3}_{x^2b_h(x)})$$

$$= (a_l(x) + x^2a_h(x)) \cdot (b_l(x) + x^2b_h(x))$$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Polynomials with 4 coefficients:

$$
\begin{aligned}
c_0 + \cdots + c_6x^6 :=& (\underbrace{a_0 + a_1x}_{a_l(x)} + \underbrace{a_2x^2 + a_3x^3}_{x^2a_h(x)}) \cdot (\underbrace{b_0 + b_1x}_{b_l(x)} + \underbrace{b_2x^2 + b_3x^3}_{x^2b_h(x)}) \\
=& (a_l(x) + x^2a_h(x)) \cdot (b_l(x) + x^2b_h(x)) \\
=& a_l \cdot b_l + \\
& [(a_l + a_h) \cdot (b_l + b_h) - a_l \cdot b_l - a_h \cdot b_h]x^2 \\
& + a_h \cdot b_h x^4
\end{aligned}
$$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Polynomials with 4 coefficients:

$$
\begin{aligned}
c_0 + \cdots + c_6x^6 := & (\underbrace{a_0 + a_1x}_{a_l(x)} + \underbrace{a_2x^2 + a_3x^3}_{x^2a_h(x)}) \cdot (\underbrace{b_0 + b_1x}_{b_l(x)} + \underbrace{b_2x^2 + b_3x^3}_{x^2b_h(x)}) \\
= & (a_l(x) + x^2a_h(x)) \cdot (b_l(x) + x^2b_h(x)) \\
= & a_l \cdot b_l + \\
& [(a_l + a_h) \cdot (b_l + b_h) - a_l \cdot b_l - a_h \cdot b_h]x^2 \\
& + a_h \cdot b_hx^4
\end{aligned}
$$

Naive $(9\times, 25+)$ vs. Karatsuba $(16\times, 12(+,-))$

Karatsuba - Is it better ?

Polynomials with 2 coefficients:

Naive $(4\times, 1+)$ vs. Karatsuba $(3\times, 4(+,-))$

Polynomials with 4 coefficients:

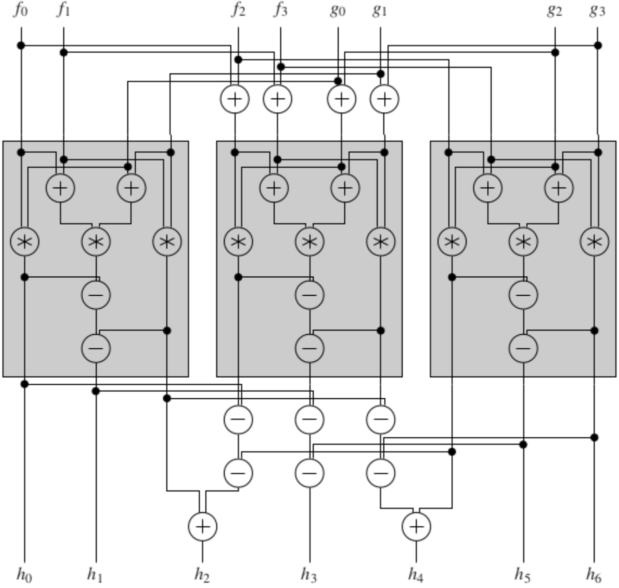
$$
\begin{aligned}
c_0 + \cdots + c_6x^6 := &(\underbrace{a_0 + a_1x}_{a_l(x)} + \underbrace{a_2x^2 + a_3x^3}_{x^2a_h(x)}) \cdot (\underbrace{b_0 + b_1x}_{b_l(x)} + \underbrace{b_2x^2 + b_3x^3}_{x^2b_h(x)}) \\
&= (a_l(x) + x^2a_h(x)) \cdot (b_l(x) + x^2b_h(x)) \\
&= a_l \cdot b_l + \\
&\quad [(a_l + a_h) \cdot (b_l + b_h) - a_l \cdot b_l - a_h \cdot b_h]x^2 \\
&\quad + a_h \cdot b_h x^4
\end{aligned}
$$

Naive $(9\times, 25+)$ vs. Karatsuba $(16\times, 12(+,-))$

Polynomials with 8 coefficients:

Naive $(64\times, 56+)$ vs. Karatsuba $(27\times, 65(+,-))$

Karatsuba - Is it better ?



Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = \sum_{0 \leq i < n} a_i x^i$ Split in 2 parts

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = \sum_{0 \leq i < n/2} a_i x^i + x^{n/2} \sum_{0 \leq i < n/2} a_{i+n/2} x^i$ Split in 2 parts

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = a_l(x) + x^{n/2} a_h(x)$ Split in 2 parts

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = a_l(x) + x^{n/2} a_h(x)$ Split in 2 parts
2. $b(x) = b_l(x) + x^{n/2} b_h(x)$

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = a_l(x) + x^{n/2} a_h(x)$ — Split in 2 parts
2. $b(x) = b_l(x) + x^{n/2} b_h(x)$
3. $c_l(x) = a_l(x) \cdot b_l(x)$ — Recursive call of size $n/2$
4. $c_m(x) = (a_l + a_h) \cdot (b_l + b_h)$ — Recursive call of size $n/2$
5. $c_h(x) = a_h(x) \cdot b_h(x)$ — Recursive call of size $n/2$

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

Recursive multiplication algorithm:

1. $a(x) = a_l(x) + x^{n/2} a_h(x)$ — Split in 2 parts
2. $b(x) = b_l(x) + x^{n/2} b_h(x)$
3. $c_l(x) = a_l(x) \cdot b_l(x)$ — Recursive call of size $n/2$
4. $c_m(x) = (a_l + a_h) \cdot (b_l + b_h)$ — Recursive call of size $n/2$
5. $c_h(x) = a_h(x) \cdot b_h(x)$ — Recursive call of size $n/2$
6. **return** $c(x) = c_l(x) + (c_m(x) - c_l(x) - c_h(x))\,x^{n/2} + c_h(x)x^n$

Karatsuba - Divide and conquer algorithm

What about multiplication $a(x) \cdot b(x)$ of polynomials of degree n?

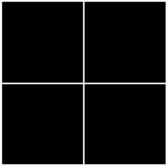
Recursive multiplication algorithm:

1. $a(x) = a_l(x) + x^{n/2} a_h(x)$ Split in 2 parts
2. $b(x) = b_l(x) + x^{n/2} b_h(x)$
3. $c_l(x) = a_l(x) \cdot b_l(x)$ Recursive call of size $n/2$
4. $c_m(x) = (a_l + a_h) \cdot (b_l + b_h)$ Recursive call of size $n/2$
5. $c_h(x) = a_h(x) \cdot b_h(x)$ Recursive call of size $n/2$
6. **return** $c(x) = c_l(x) + (c_m(x) - c_l(x) - c_h(x))\, x^{n/2} + c_h(x) x^n$

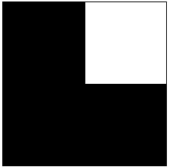
Remarks:

- Complexity: $K(n) = 3K(n/2) + O(n) = O(n^{\log_2(3)}) = O(n^{1,59})$
- Karatsuba $K(n) \ll O(n^2)$ naive
- In practice, hybrid Karatsuba / naive algorithm
- Need careful memory management (one memory allocation, in-place algorithms)

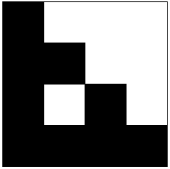
Karatsuba - Complexity



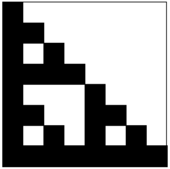
classical



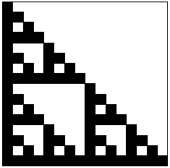
1 iteration



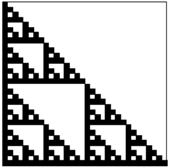
2 iterations



3 iterations



4 iterations



5 iterations

Outline

Polynomial multiplication algorithms:

1. Karatsuba
2. **Fast Fourier Tranform (FFT)**
3. Truncated FFT (TFT)

Evaluation / Interpolation algorithms

Monomial representation		Evaluation representation
$\begin{cases} a(x) \text{ degree } n \\ b(x) \text{ degree } n \end{cases}$	$\xrightarrow{\textit{Evaluation}}$	$\begin{cases} a(0), a(1), \ldots, a(2n+1) \\ b(0), b(1), \ldots, b(2n+1) \end{cases}$
$\Big\downarrow$ *Mult.*		*Pointwise mult.* $\Big\downarrow$
$c(x) = a(x) \cdot b(x)$ of degree $2n$	$\xleftarrow{\textit{Interpolation}}$	$\begin{cases} c(0) = a(0) \cdot b(0) \\ \quad \ldots \\ c(2n+1) = a(2n+1) \cdot b(2n+1) \end{cases}$

Evaluation / Interpolation algorithms

Monomial representation		Evaluation representation
$\begin{cases} a(x) \text{ degree } n \\ b(x) \text{ degree } n \end{cases}$	$\xrightarrow{\textit{Evaluation}}$	$\begin{cases} a(0), a(1), \dots, a(2n+1) \\ b(0), b(1), \dots, b(2n+1) \end{cases}$
$\downarrow$ *Mult.*		*Pointwise mult.* $\downarrow$ Cost : $O(n)$
$c(x) = a(x) \cdot b(x)$ of degree $2n$	$\xleftarrow{\textit{Interpolation}}$	$\begin{cases} c(0) = a(0) \cdot b(0) \\ \dots \\ c(2n+1) = a(2n+1) \cdot b(2n+1) \end{cases}$

Evaluation / Interpolation algorithms

Monomial representation		Evaluation representation
$\begin{cases} a(x) \text{ degree } n \\ b(x) \text{ degree } n \end{cases}$	$\xrightarrow[\textit{Cost?}]{\textit{Evaluation}}$	$\begin{cases} a(0), a(1), \ldots, a(2n+1) \\ b(0), b(1), \ldots, b(2n+1) \end{cases}$
$\downarrow$ *Mult.*		*Pointwise mult.* $\downarrow$ Cost : $O(n)$
$c(x) = a(x) \cdot b(x)$ of degree $2n$	$\xleftarrow[\textit{Cost?}]{\textit{Interpolation}}$	$\begin{cases} c(0) = a(0) \cdot b(0) \\ \ldots \\ c(2n+1) = a(2n+1) \cdot b(2n+1) \end{cases}$

Evaluation / Interpolation algorithms

Monomial representation		Evaluation representation
$\begin{cases} a(x) \text{ degree } n \\ b(x) \text{ degree } n \end{cases}$	$\xrightarrow[\text{Cost?}]{\text{Evaluation}}$	$\begin{cases} a(0), a(1), \ldots, a(2n+1) \\ b(0), b(1), \ldots, b(2n+1) \end{cases}$
$\big\downarrow$ *Mult.*		Pointwise mult. $\big\downarrow$ Cost : $O(n)$
$c(x) = a(x) \cdot b(x)$ of degree $2n$	$\xleftarrow[\text{Cost?}]{\text{Interpolation}}$	$\begin{cases} c(0) = a(0) \cdot b(0) \\ \ldots \\ c(2n+1) = a(2n+1) \cdot b(2n+1) \end{cases}$

From now on, we will focus on the cost of evaluation / interpolation.

Evaluation / Interpolation algorithms

Monomial representation		Evaluation representation
$\begin{cases} a(x) \text{ degree } n \\ b(x) \text{ degree } n \end{cases}$	$\xrightarrow[\color{red}{Cost?}]{Evaluation}$	$\begin{cases} a(0), a(1), \dots, a(2n+1) \\ b(0), b(1), \dots, b(2n+1) \end{cases}$
$\downarrow$ *Mult.*		*Pointwise mult.* $\downarrow$ $\color{red}{Cost : O(n)}$
$c(x) = a(x) \cdot b(x)$ of degree $2n$	$\xleftarrow[\color{red}{Cost?}]{Interpolation}$	$\begin{cases} c(0) = a(0) \cdot b(0) \\ \dots \\ c(2n+1) = a(2n+1) \cdot b(2n+1) \end{cases}$

From now on, we will focus on the cost of evaluation / interpolation.

Related interesting problem: Interpolation with errors.

Discrete Fourier Transform (DFT)

Evaluation / interpolation is generally costly.

But if evaluation points are specific, it can be *very efficient*:

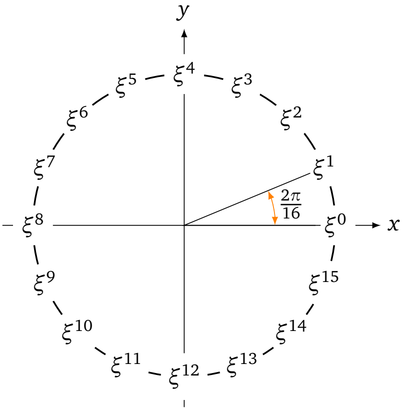
- Evaluate at $\xi^0, \xi^1, \xi^2, \ldots, \xi^{n-1}$
- where ξ is a primitive root of unity

Discrete Fourier Transform = Evaluation on roots of unity

$$DFT_\xi(a(x)) := (a(\xi^0), \ldots, a(\xi^{n-1}))$$

where ξ is a n-th *primitive root of unity* and $\deg a(x) < n$.

Complex root of unity



$\xi = e^{\frac{2i\pi}{16}} \in \mathbb{C}$ is a 16-*th primitive root of unity*:

- $\xi^{16} = 1$
- $\xi^i \neq 1$ for $0 < i < 16$

Remark:

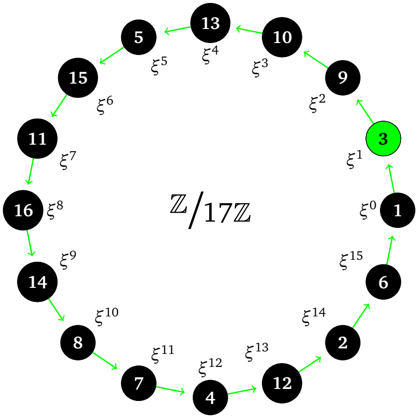
- $1 = \xi^0 = \xi^{16} = \xi^{32} = \dots$
- $\xi^{16/2} = -1$

Pros & Cons: Fast floating point arithmetic, but precision issues

Modular root of unity

Modular integers: $\mathbb{Z}/p\mathbb{Z}$ if p prime $\quad (a = a + p = a + 2p = \dots \text{ modulo } p)$

Example $\mathbb{Z}/17\mathbb{Z}$**:**



$\xi = 3 \in \mathbb{Z}/17\mathbb{Z}$ is a *16-th primitive root of unity*:

- $\xi^{16} = 1$
- $\xi^i \neq 1$ for $0 < i < 16$

Remark:

- $1 = \xi^0 = \xi^{16} = \xi^{32} = \dots$
- $\xi^{16/2} = -1$

Fast Fourier Transform

Goal: Given $a(x)$**, compute** $(a(\xi^0), \ldots, a(\xi^{n-1}))$ **(when** $n = 2^k$**)**

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$a(\xi^j) = a_l(\xi^j) + (\xi^j)^{n/2} a_h(\xi^j)$

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0), \ldots, a(\xi^{n-1}))$ (when $n = 2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (\xi^{n/2})^j a_h(\xi^j)$$

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0), \ldots, a(\xi^{n-1}))$ (when $n = 2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j)$$

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0), \ldots, a(\xi^{n-1}))$ (when $n = 2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j) \quad \Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) \end{cases}$$

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0), \ldots, a(\xi^{n-1}))$ (when $n = 2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j) \;\Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) \end{cases}$$

Define $\bar{r}(x) = a_l(x) + a_h(x)$, $\underline{r}'(x) = a_l(x) - a_h(x)$.

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0),\ldots,a(\xi^{n-1}))$ (when $n=2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j) \quad \Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) = \bar{r}(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) = \underline{r}'(\xi^{2i+1}) \end{cases}$$

Define $\bar{r}(x) = a_l(x) + a_h(x)$, $\underline{r}'(x) = a_l(x) - a_h(x)$.

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0),\dots,a(\xi^{n-1}))$ (when $n=2^k$)

If $a(x)=a_l(x)+x^{n/2}a_h(x)$ then

$$a(\xi^j)=a_l(\xi^j)+(-1)^j a_h(\xi^j) \quad \Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) = \bar{r}(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) = \underline{r}'(\xi^{2i+1}) \end{cases}$$

Define $\bar{r}(x)=a_l(x)+a_h(x)$, $\underline{r}'(x)=a_l(x)-a_h(x)$ and $\underline{r}(x)=\underline{r}'(\xi x)$.

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0),\dots,a(\xi^{n-1}))$ (when $n=2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j) \quad \Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) = \bar{r}(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) = \underline{r}(\xi^{2i}) \end{cases}$$

Define $\bar{r}(x) = a_l(x) + a_h(x)$, $\underline{r}'(x) = a_l(x) - a_h(x)$ and $\underline{r}(x) = \underline{r}'(\xi x)$.

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0), \ldots, a(\xi^{n-1}))$ (when $n = 2^k$)

If $a(x) = a_l(x) + x^{n/2} a_h(x)$ then

$$a(\xi^j) = a_l(\xi^j) + (-1)^j a_h(\xi^j) \quad \Rightarrow \begin{cases} a(\xi^{2i}) = a_l(\xi^{2i}) + a_h(\xi^{2i}) = \bar{r}(\xi^{2i}) \\ a(\xi^{2i+1}) = a_l(\xi^{2i+1}) - a_h(\xi^{2i+1}) = \underline{r}(\xi^{2i}) \end{cases}$$

Define $\bar{r}(x) = a_l(x) + a_h(x)$, $\underline{r}'(x) = a_l(x) - a_h(x)$ and $\underline{r}(x) = \underline{r}'(\xi x)$.

Finally $(a(\xi^0), a(\xi^1), a(\xi^2), a(\xi^3), \ldots) = (\bar{r}(\xi^0), \underline{r}(\xi^0), \bar{r}(\xi^2), \underline{r}(\xi^2), \ldots)$

Fast Fourier Transform

Goal: Given $a(x)$, compute $(a(\xi^0),\dots,a(\xi^{n-1}))$ (when $n=2^k$)

If $a(x)=a_l(x)+x^{n/2}a_h(x)$ then

$$a(\xi^j)=a_l(\xi^j)+(-1)^j a_h(\xi^j) \quad\Rightarrow \begin{cases} a(\xi^{2i})=a_l(\xi^{2i})+a_h(\xi^{2i})=\bar{r}(\xi^{2i}) \\ a(\xi^{2i+1})=a_l(\xi^{2i+1})-a_h(\xi^{2i+1})=\underline{r}(\xi^{2i}) \end{cases}$$

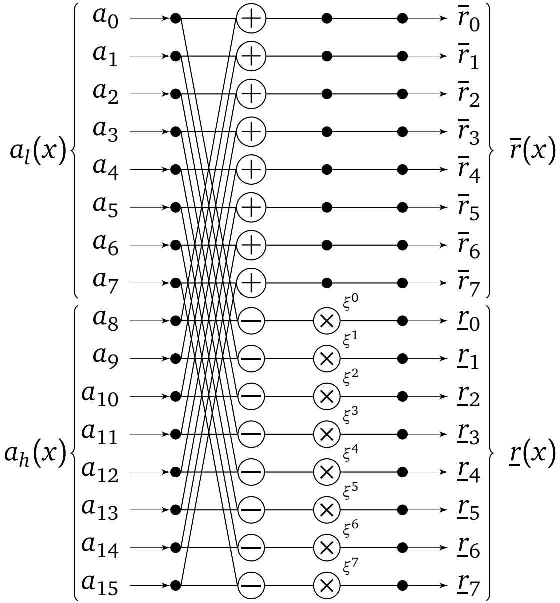
Define $\bar{r}(x)=a_l(x)+a_h(x)$, $\underline{r}'(x)=a_l(x)-a_h(x)$ and $\underline{r}(x)=\underline{r}'(\xi x)$.

Finally $(a(\xi^0),a(\xi^1),a(\xi^2),a(\xi^3),\dots)=(\bar{r}(\xi^0),\underline{r}(\xi^0),\bar{r}(\xi^2),\underline{r}(\xi^2),\dots)$

FFT Algorithm: [Cooley, Tukey '65]

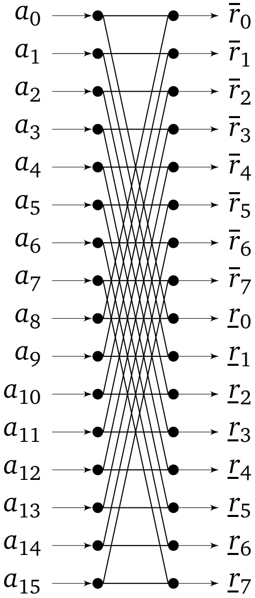
1. Write $a(x)=a_l(x)+x^{n/2}a_h(x)$ — Split in 2 parts
2. Compute $\bar{r}(x)=a_l(x)+a_h(x)$
3. Compute $\underline{r}'(x)=a_l(x)-a_h(x)$
4. Compute $\underline{r}(x)=\underline{r}'(\xi x)$
5. Evaluate $\bar{r}(\xi^0),\bar{r}(\xi^2),\bar{r}(\xi^4),\dots$ — Recursive call in size $n/2$
6. Evaluate $\underline{r}(\xi^0),\underline{r}(\xi^2),\underline{r}(\xi^4),\dots$ — Recursive call in size $n/2$
7. **return** $\bar{r}(\xi^0),\underline{r}(\xi^0),\bar{r}(\xi^2),\underline{r}(\xi^2),\bar{r}(\xi^4),\underline{r}(\xi^4),\dots$

FFT butterflies

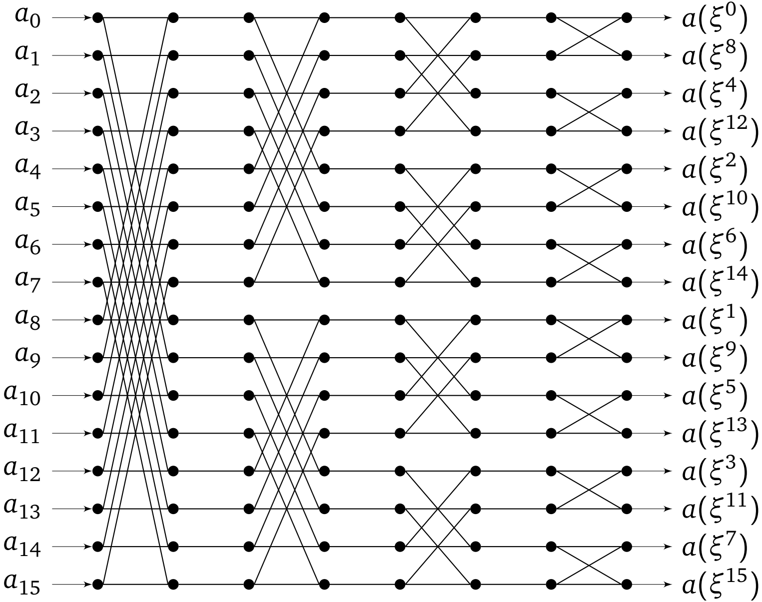


- $\bar{r}(x) = a_l(x) + a_h(x)$
- $\underline{r}'(x) = a_l(x) - a_h(x)$
- $\underline{r}(x) = \underline{r}'(\xi x)$

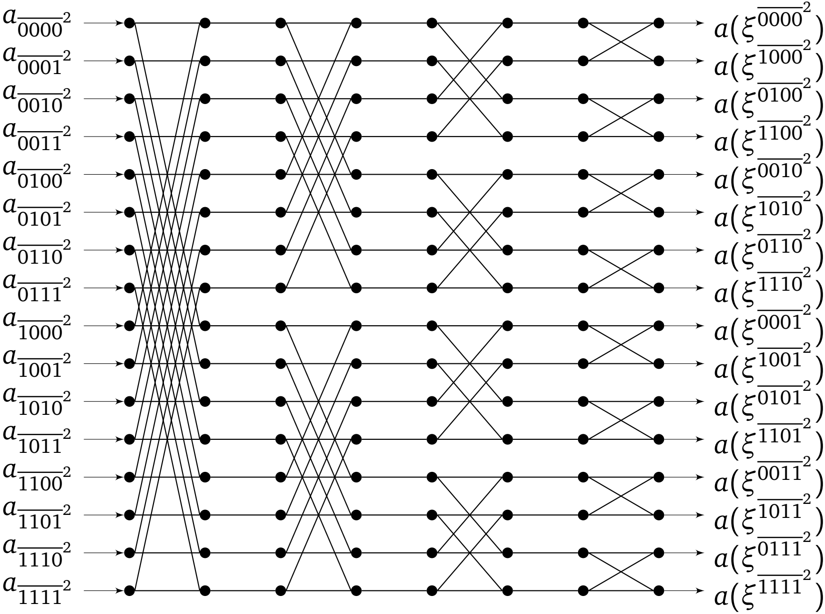
FFT butterflies



FFT butterflies



FFT butterflies



FFT timings

- Evaluation or interpolation in $3/2n\log n$ arithmetic operations
- Multiplication in $\sim 9n\log n$
- But **only for degree** $n = 2^k$, pad with zeroes otherwise, loose factor 2

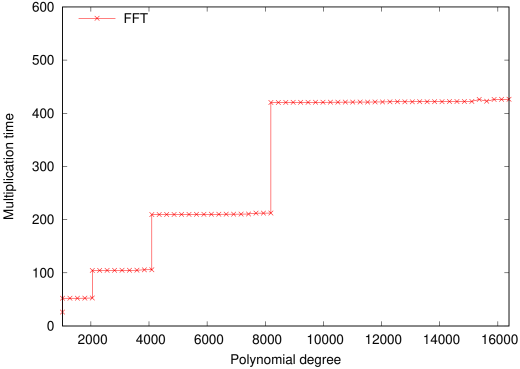


Figure 1: Fast Fourier Transform timings

Outline

Polynomial multiplication algorithms:

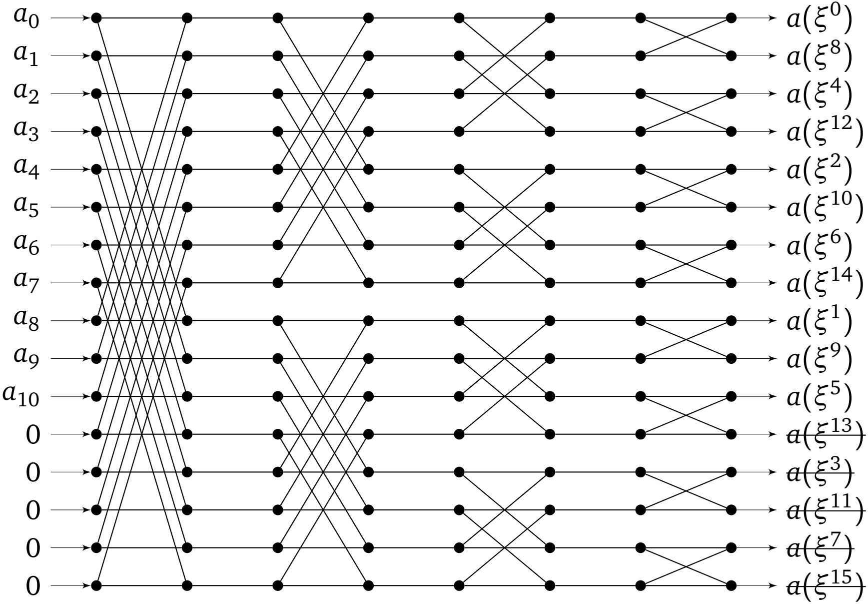
1. Karatsuba
2. Fast Fourier Tranform (FFT)
3. **Truncated FFT (TFT)**

Truncated Fourier Transform [Hoeven '04]

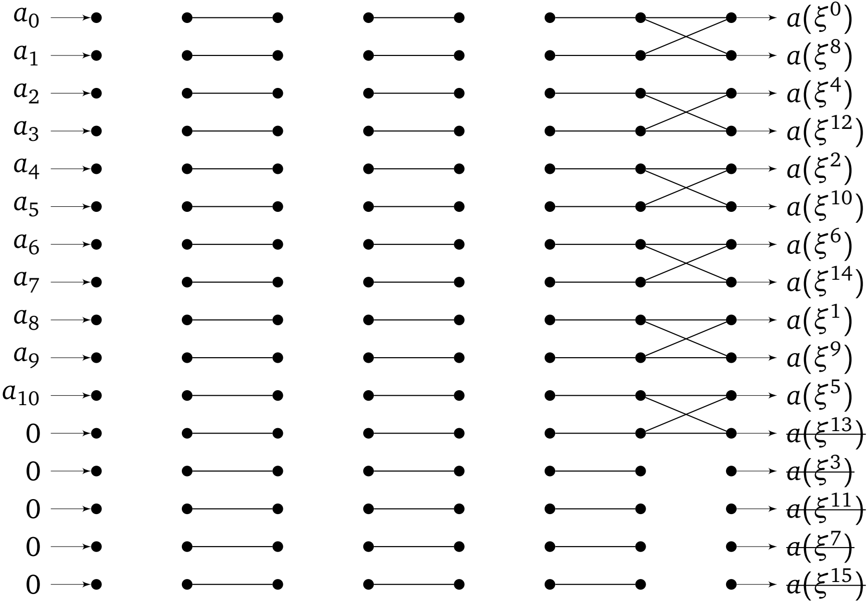
Goal: Save computations when $n = \deg a(x)$ is not 2^k:

- compute only the first n evaluates of $a(x)$
- get a cost $\sim \frac{3}{2} n \log n$ for **all degrees** n (instead of $\sim \frac{3}{2} 2^k \log 2^k$)
- save up to a factor 2

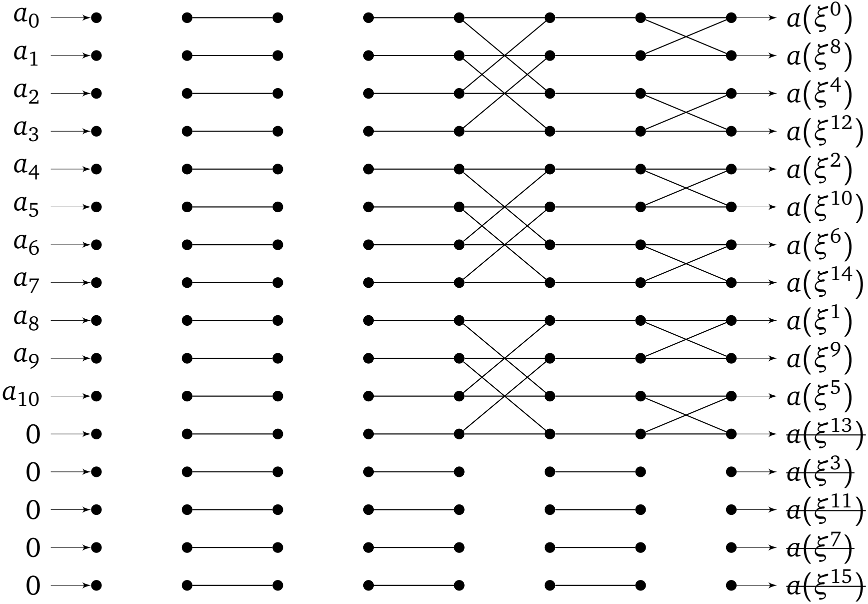
Truncated Fourier Transform



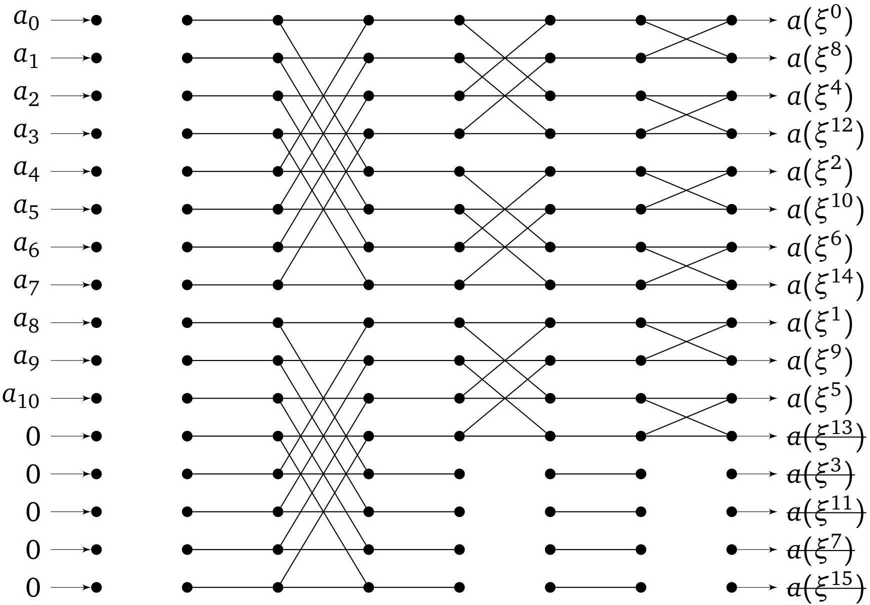
Truncated Fourier Transform



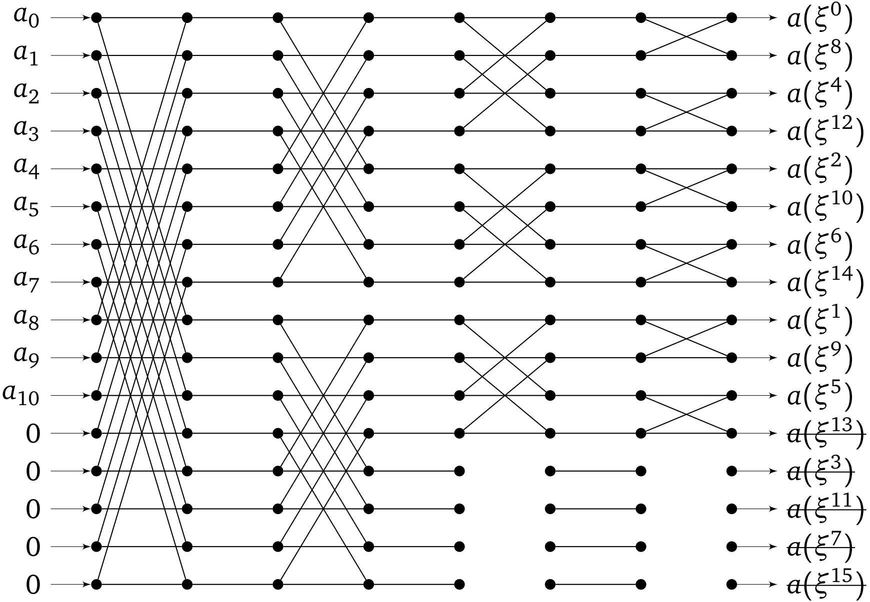
Truncated Fourier Transform



Truncated Fourier Transform



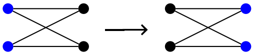
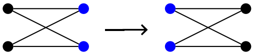
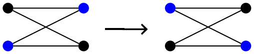
Truncated Fourier Transform



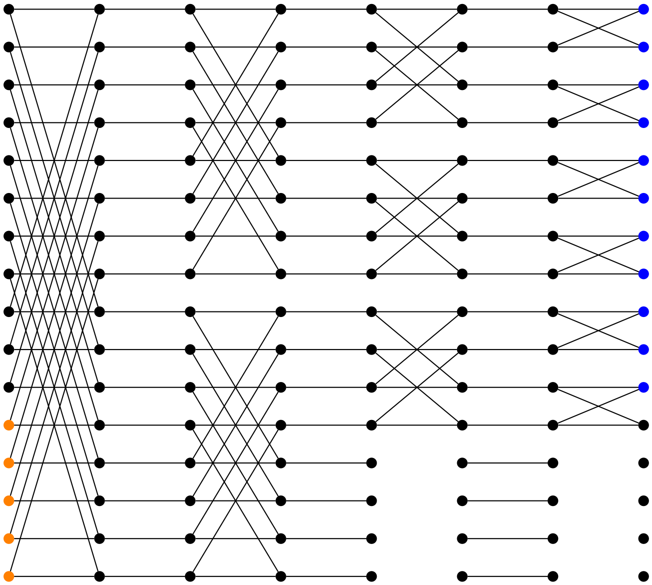
Goal:

- recover the polynomial $a(x)$ from only its first n evaluates
- knowing that $\deg a(x) < n$ (instead of $\deg a(x) < 2^k$)
- get a cost $\sim \frac{3}{2} n \log n$ (instead of $\sim \frac{3}{2} 2^k \log 2^k$)
- save up to a factor 2

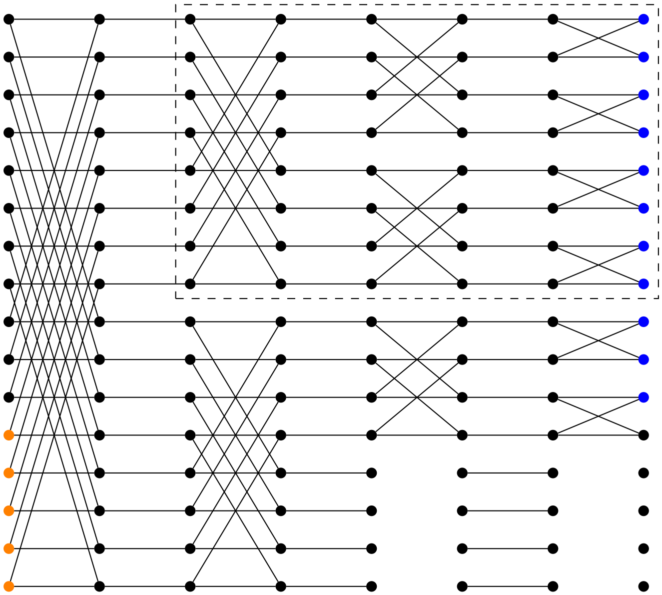
Operations required:

- FFT butterfly: 
- Inverse FFT butterfly: 
- Crossed butterfly: 

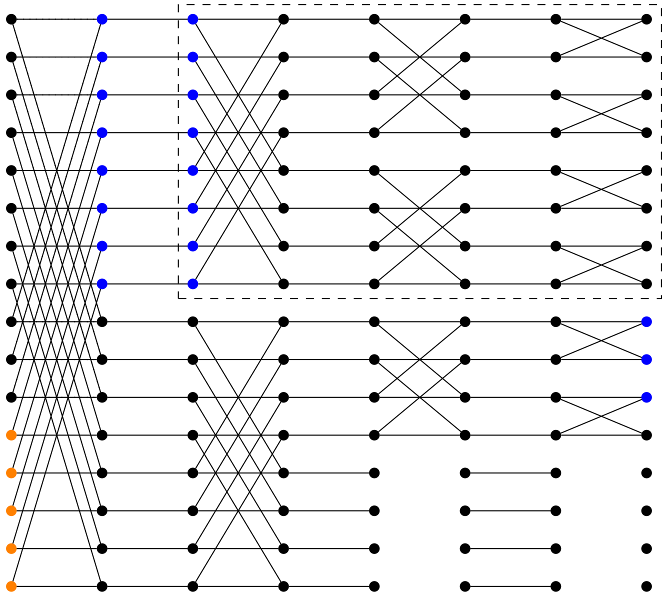
Inverse TFT - Recursive Algorithm: Case 1



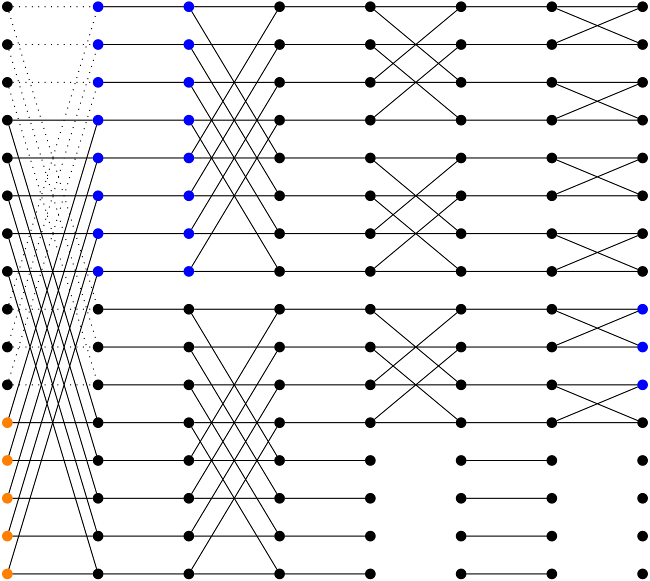
Inverse TFT - Recursive Algorithm: Case 1



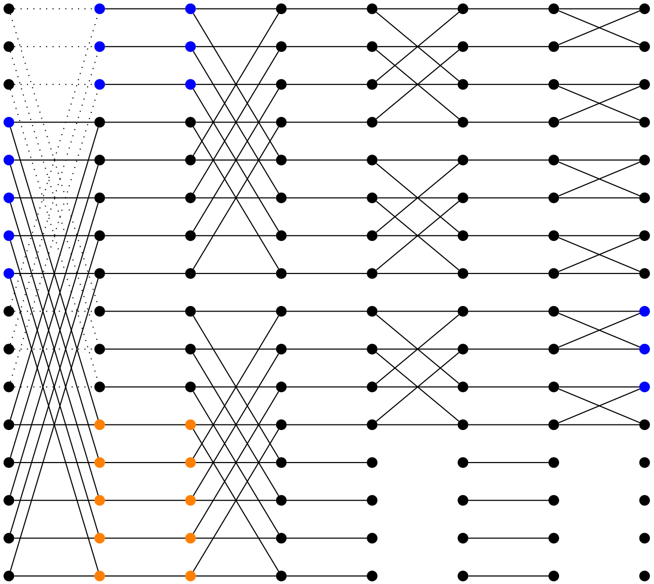
Inverse TFT - Recursive Algorithm: Case 1



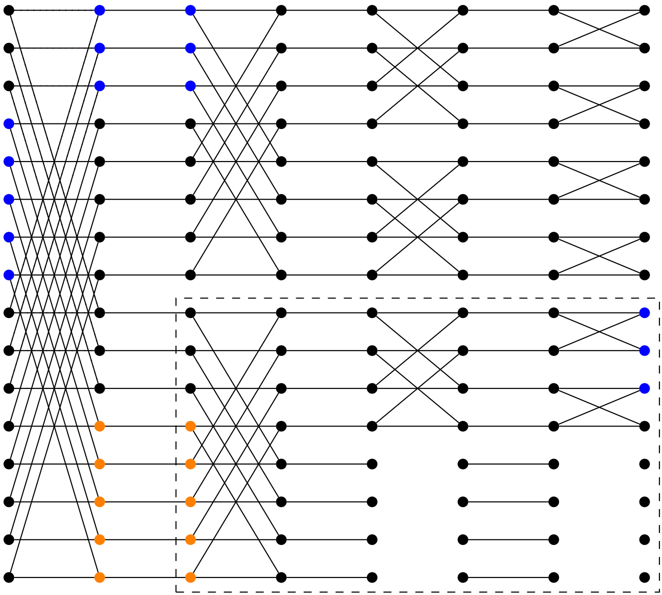
Inverse TFT - Recursive Algorithm: Case 1



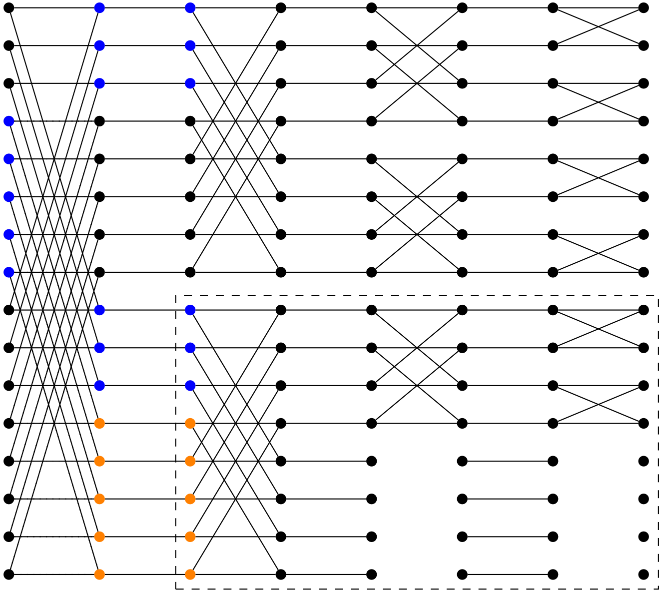
Inverse TFT - Recursive Algorithm: Case 1



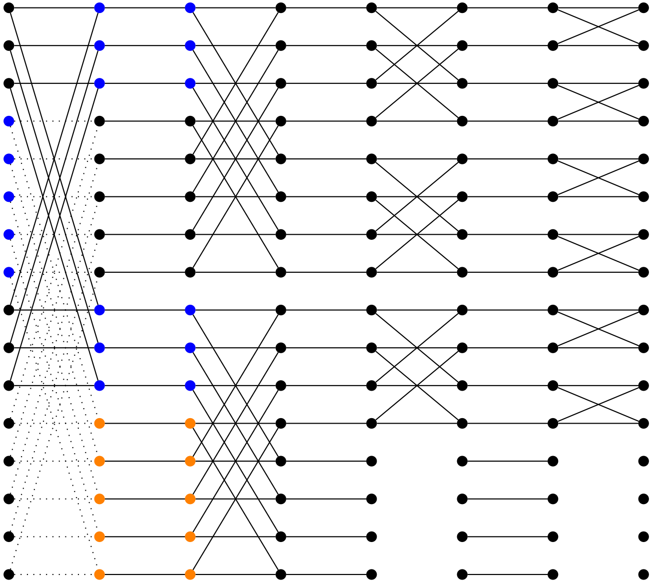
Inverse TFT - Recursive Algorithm: Case 1



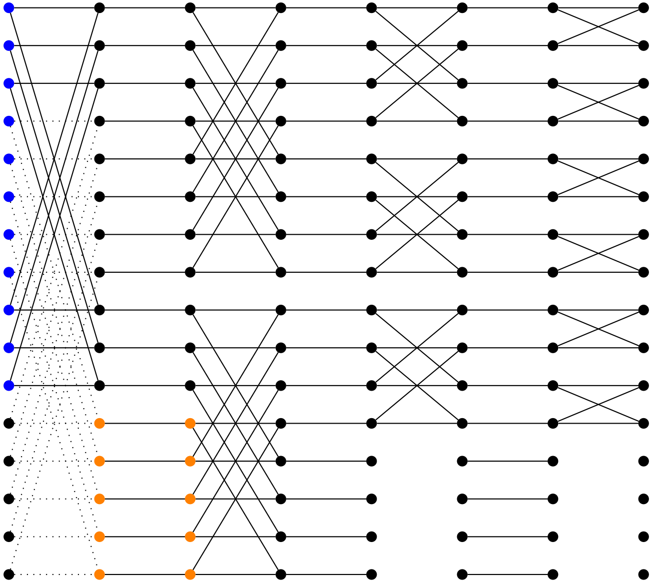
Inverse TFT - Recursive Algorithm: Case 1



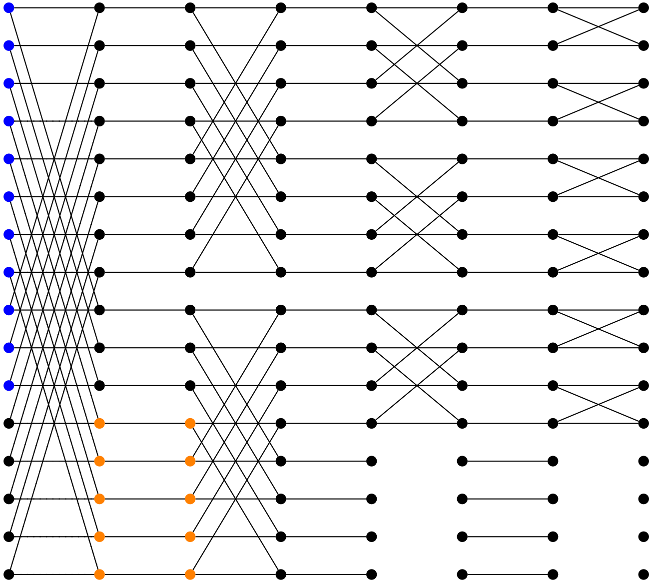
Inverse TFT - Recursive Algorithm: Case 1



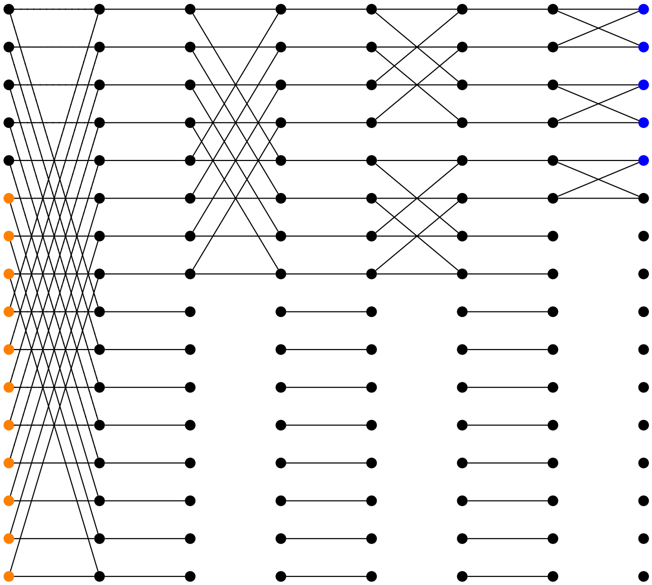
Inverse TFT - Recursive Algorithm: Case 1



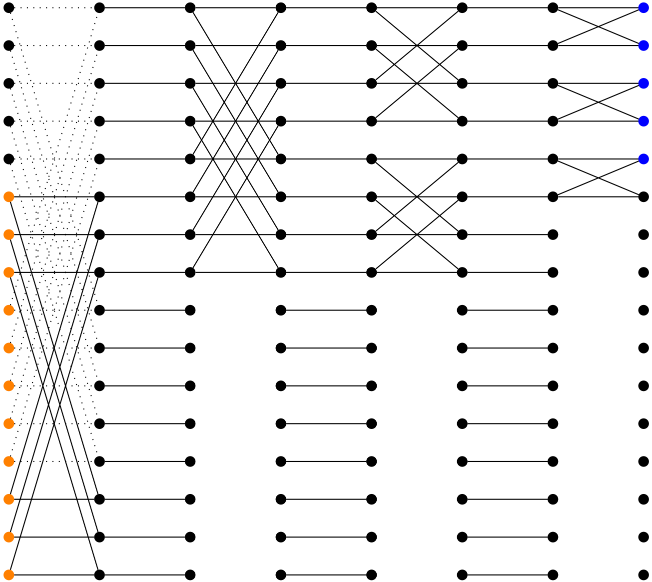
Inverse TFT - Recursive Algorithm: Case 1



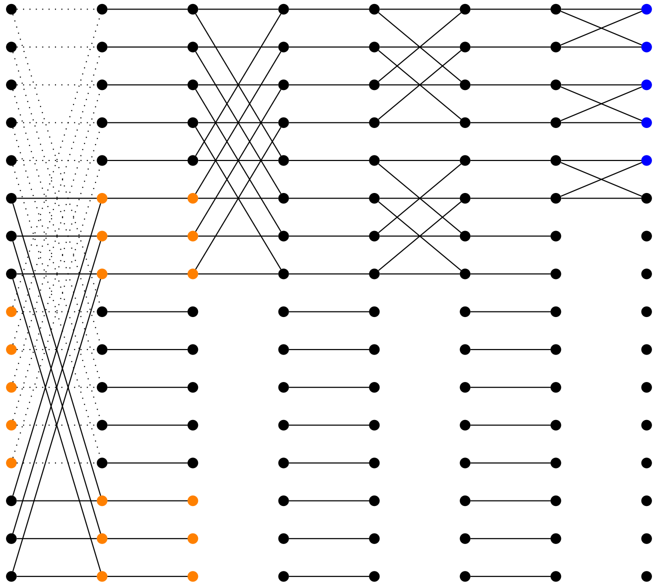
Inverse TFT - Recursive Algorithm: Case 2



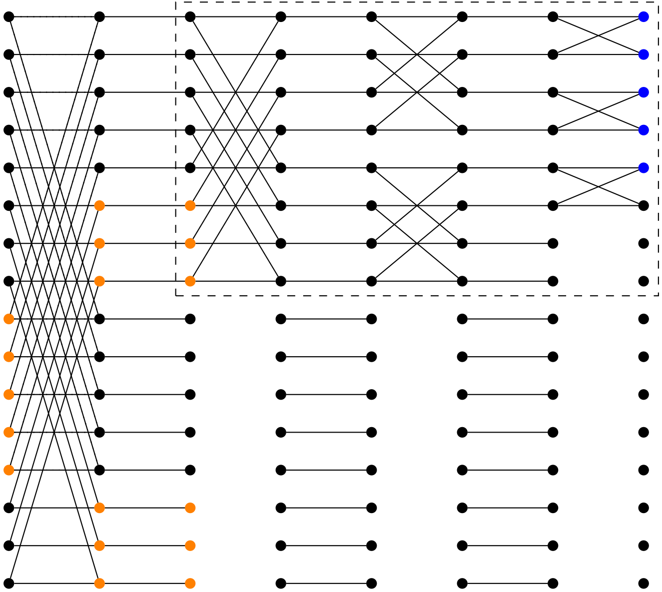
Inverse TFT - Recursive Algorithm: Case 2



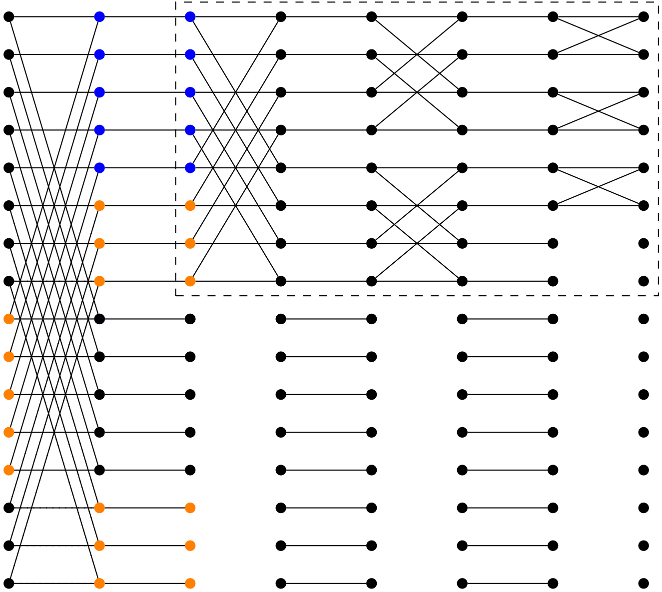
Inverse TFT - Recursive Algorithm: Case 2



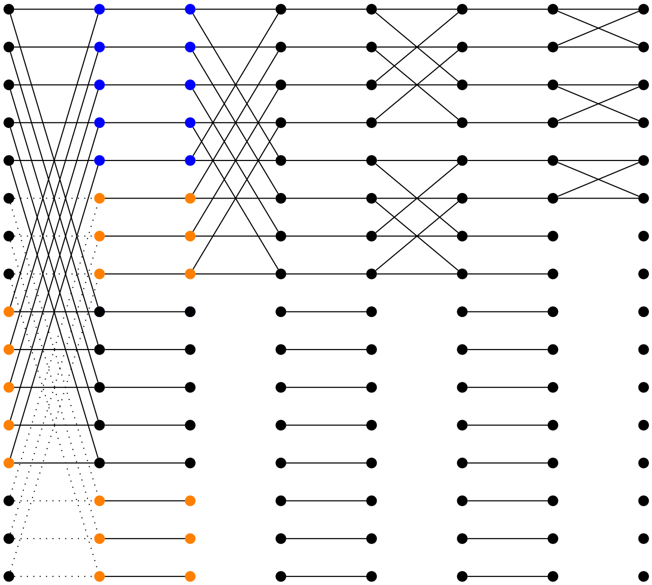
Inverse TFT - Recursive Algorithm: Case 2



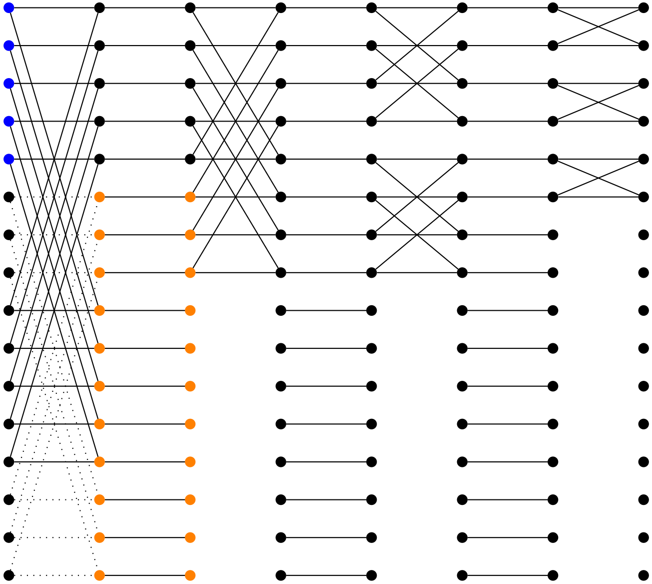
Inverse TFT - Recursive Algorithm: Case 2



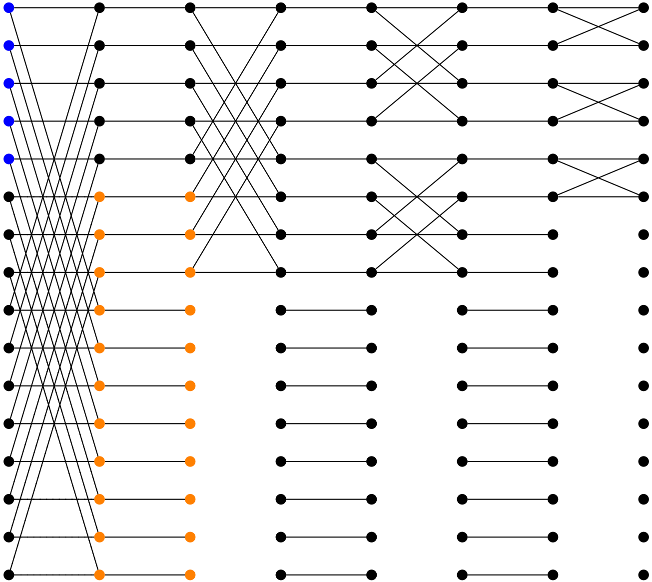
Inverse TFT - Recursive Algorithm: Case 2



Inverse TFT - Recursive Algorithm: Case 2



Inverse TFT - Recursive Algorithm: Case 2



Complexity and timings

Evaluation or interpolation in $3/2n\log n$ for **all degrees** n

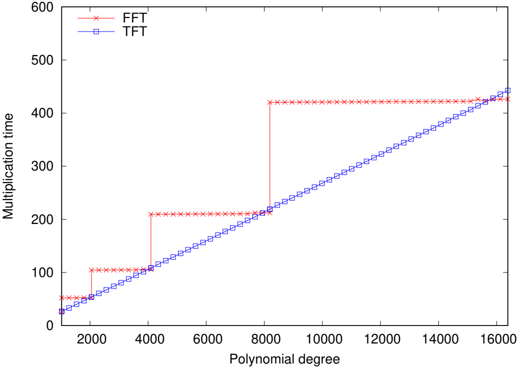
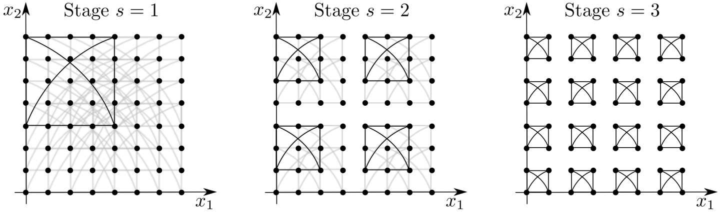


Figure 2: Fast Fourier Transform vs Truncated FFT timings

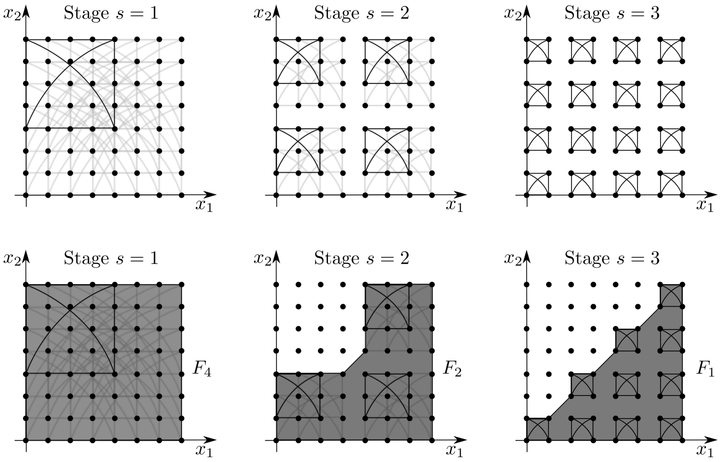
Link with team research

- Implementation of polynomial matrix multiplication in LinBox
- FFT for lattice and symmetric polynomials [Hoeven, L., Schost '14]



Link with team research

- Implementation of polynomial matrix multiplication in LinBox
- FFT for lattice and symmetric polynomials [Hoeven, L., Schost '14]



Conclusion

- Open questions on multiplication of polynomials of degree n:

Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$

Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log\log n)$ in general [Schönhage, Strassen '71]

Conclusion

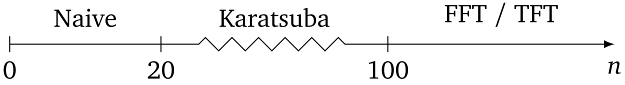
- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log \log n)$ in general [Schönhage, Strassen '71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf '17]

Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log \log n)$ in general [Schönhage, Strassen '71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf '17]
- In practice:

Conclusion

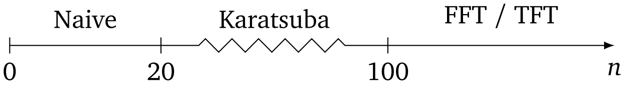
- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log \log n)$ in general [Schönhage, Strassen '71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf '17]
- In practice:
  - Complementary algorithms:



Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log \log n)$ in general [Schönhage, Strassen '71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf '17]

- In practice:
  - Complementary algorithms:

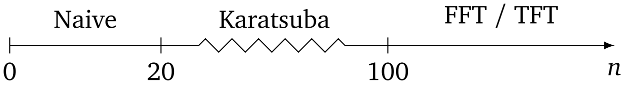


  - Will [HHL '17] become practical in the future ?

Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log\log n)$ in general [Schönhage, Strassen '71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf '17]

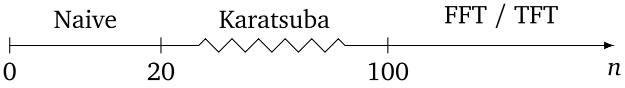
- In practice:
  - Complementary algorithms:



  - Will [HHL '17] become practical in the future ?
  - Technical aspects not discussed: SIMD, cache, …

Conclusion

- Open questions on multiplication of polynomials of degree n:
  - Complexity $O(n \log n)$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ but $n < p$
  - Complexity $O(n \log n \log\log n)$ in general [Schönhage, Strassen ’71]
  - Complexity $O(n \log n 8^{\log^* n})$ when coefficients in $\mathbb{Z}/p\mathbb{Z}$ [Harvey, Hoeven, Lecerf ’17]
- In practice:
  - Complementary algorithms:



  - Will [HHL ’17] become practical in the future ?
  - Technical aspects not discussed: SIMD, cache, ...
- Thank you for your attention !

Different Fourier transforms

Classical Fourier transform

- Decomposition in the frequency domain
- Integral formula:

$$\hat{f}(\xi) = \int f(x) e^{-2i\pi x\xi} dx$$

- Multiplicativity: $\hat{h}(\xi) = \hat{f}(\xi) \cdot \hat{g}(\xi)$ when $h = \text{Convolution}(f, g)$

Discrete Fourier transform

- Discrete formula:

$$\hat{f}_k = \sum_n f_n e^{-\frac{2i\pi}{N} nk}$$

- Link with evaluation: $\hat{p}_k = P(e^{-\frac{2i\pi}{N} k})$ where $P(x) = \sum_n p_n x^n$
- Multiplicativity: Let $c(x) = a(x) \cdot b(x)$ then $\hat{c}_k = \hat{a}_k \cdot \hat{b}_k$