Fast Fourier Transform

Irina Bobkova

Overview

I. Polynomials

II. The DFT and FFT

III. Efficient implementations

IV. Some problems

Representation of polynomials

A polynomial in the variable x over an algebraic field F is representation of a function A(x) as a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

•<u>Coefficient representation</u>

$$a = (a_0, a_1, \ldots a_{n-1})$$

•<u>Point-value representation</u>

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

	Coefficient representation	Point-value representation
Adding	$\Theta(n)$	$\Theta(n)$
Multiplication	$\Theta(n^2)$	$\Theta(n)$

Interpolation

Interpolation-the inverse of evaluation –determining the coefficient form from a point-value representation

Lagrange's formula

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

The coefficients can be computed in time $\Theta(n^2)$

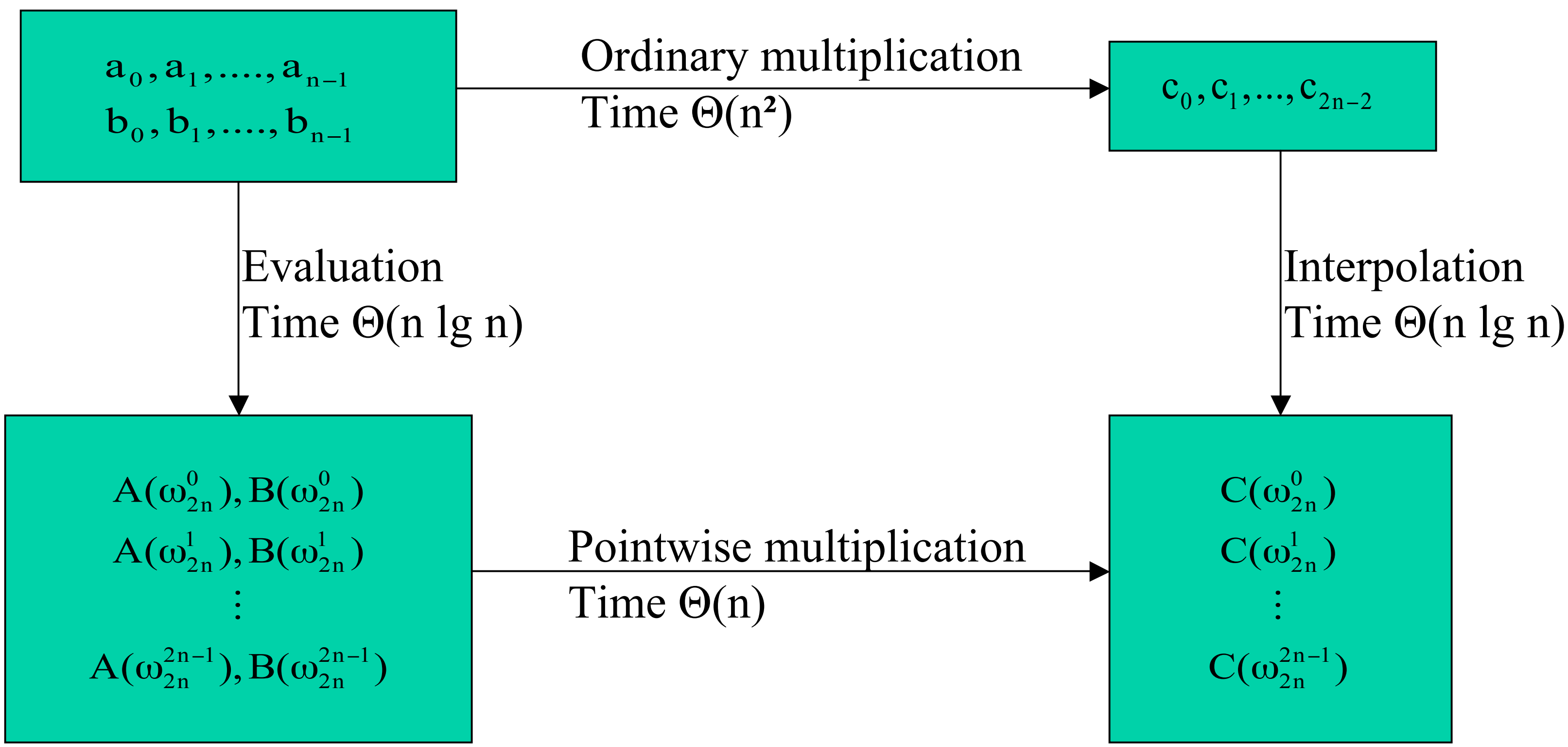
Exercise. Prove it.

Thus, n-point evaluation and interpolation are well-defined inverse operations between two representations. The algorithms described above for these problems take time $\Theta(n^2)$.

Fast multiplication

Question. Can we use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form?

Answer. Yes, but we are to be able to convert quickly from one form to another.



Complex roots of unity

$$Z^n - 1 = 0$$

There are exactly n complex roots of unity.They form a cyclic multiplication group:

$$\omega_k = e^{2\pi ik/n}$$

The value $\omega_1 = e^{2\pi i/n}$ is called **the primitive root of unity**; all of the other complex roots are powers of it.

Discrete Fourier Transform

Let F(x) be the polynomial $F(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0$ with degree-bound n, which is a power of 2.ω is a primitive n-th root of unity.

Let $y_k = F(\omega^k)$ .Then

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} * \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The vector $y = (y_0, y_1, \ldots y_{n-1})$ is called the *Discrete Fourier Transform* of vector a. The matrix is denoted by $F_n(\omega)$.

How to find F_n^{-1}?

Proposition. Let ω be a primitive *l*-th root of unity over a field L.Then

$$\sum_{k=0}^{l-1} \omega^k = \begin{Bmatrix} 0 & if\, l > 1 \\ 1 & otherwise \end{Bmatrix}$$

Proof. The $l = 1$ case is immediate since ω=1.

Since ω is a primitive *l*-th root, each ω^k ,$k \neq 0$ is a distinct *l*-th root of unity.

$$Z^l - 1 = (Z - \omega_l^0)(Z - \omega_l)(Z - \omega_l^2)\ldots(Z - \omega_l^{l-1}) =$$

$$= Z^l - (\sum_{k=0}^{l-1} \omega_l^k) Z^{l-1} + \ldots + (-1)^l \prod_{k=0}^{l-1} \omega_l^k$$

Comparing the coefficients of Z^{l-1} on the left and right hand sides of this equation proves the proposition.

Inverse matrix to F_n

Proposition. Let ω be an n-th root of unity. Then,

$$F_n(\omega)\cdot F_n(\omega^{-1}) = nE_n$$

Proof. The ij^{th} element of $F_n(\omega)F_n(\omega^{-1})$ is

$$\sum_{k=0}^{n-1}\omega^{ik}\omega^{-ik} = \sum_{k=0}^{n-1}\omega^{k(i-j)} = \begin{Bmatrix} 0, & \text{if } i \neq j \\ n, & \text{otherwise} \end{Bmatrix}$$

The $i=j$ case is obvious. If $i \neq j$ then ω^{i-j} will be a primitive root of unity of order l, where $l|n$. Applying the previous proposition completes the proof.

So, $F_n^{-1}(\omega) = \frac{1}{n}F_n(\omega^{-1})$

Evaluating	$\mathbf{y} = F_n(\omega)\,\mathbf{a}$
Interpolation	$\mathbf{a} = \frac{1}{n}F_n(\omega^{-1})\,\mathbf{y}$

Fast Fourier Transform

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$$

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

So, the problem of evaluating A(x) reduces to:

1. Evaluating the degree-bound n/2 polynomials

$$A^{[0]}(x) \quad \text{and} \quad A^{[1]}(x)$$

2. Combining the results

Recursive FFT

$n \leftarrow \text{length}[a]$
if $n=1$
    then return a
$\omega_n \leftarrow e^{2\pi i/n}$
$\omega \leftarrow 1$
$a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$
$a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$
$y^{[0]} \leftarrow \text{Recursive-FFT}(a^{[0]})$
$y^{[1]} \leftarrow \text{Recursive-FFT}(a^{[1]})$
for $k \leftarrow 0$ to $n/2-1$
        do $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
            $y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
            $\omega \leftarrow \omega\omega_n$
return y

Time of the Recursive-FFT

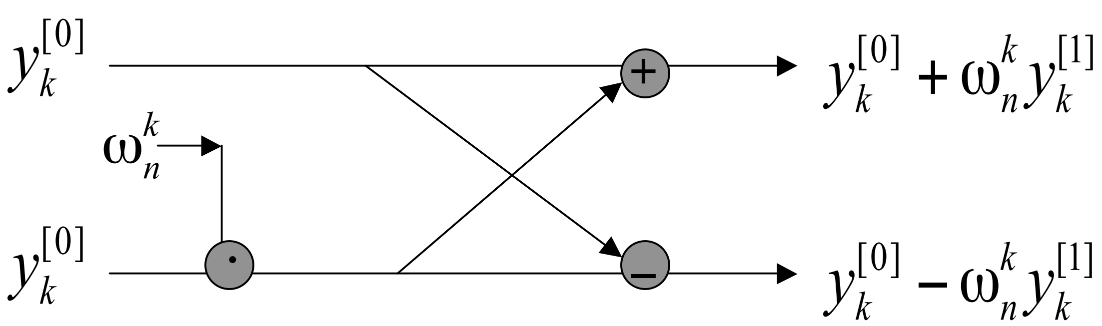
To determine the running time of procedure Recursive-FFT, we note, that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector.The recurrence for the running time is therefore

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

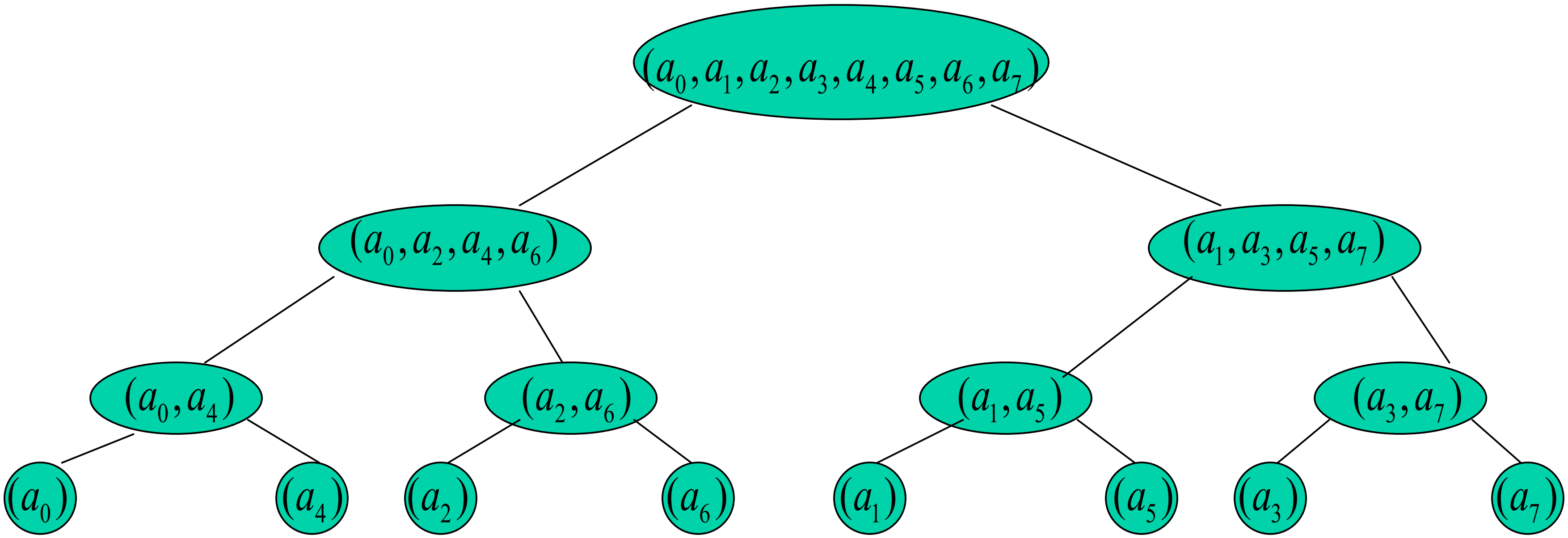
More effective implementations

The **for** loop involves computing the value $\omega_n^k y_k^{[1]}$ twice.We can change the loop(the butterfly operation):

$$
\begin{aligned}
&\textit{for}\ \ k \leftarrow 0 \text{ to } n/2\text{-}1 \\
&\quad \textit{do } t \leftarrow \omega y_k^{[1]} \\
&\qquad y_k \leftarrow y_k^{[0]} + t \\
&\qquad y_{k+(n/2)} \leftarrow y_k^{[0]} - t \\
&\qquad \omega \leftarrow \omega\omega_n
\end{aligned}
$$



Iterative FFT



1) We take the elements in pairs, compute the DFT of each pair, using one butterfly operation, and replace the pair with its DFT
2) We take these n/2 DFT's in pairs and compute the DFT of the four vector elements

.

.

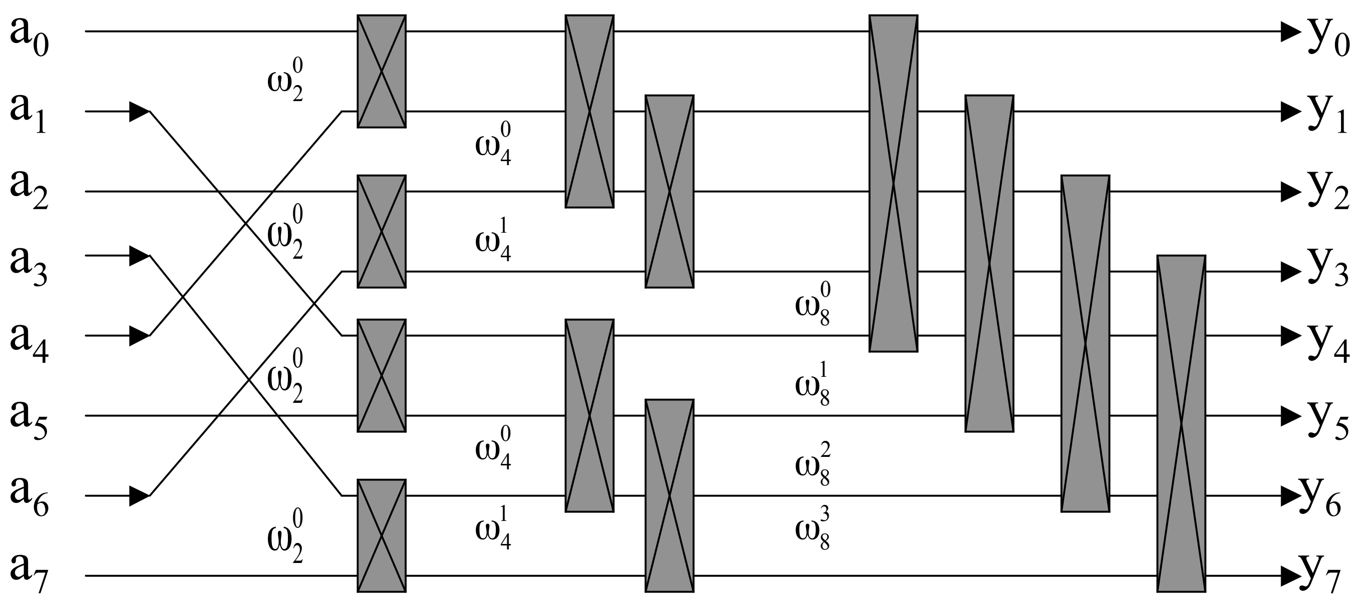
$\log_2 n$) We take 2 (n/2)-element DFT's and combine them using n/2 butterfly operations into the final n-element DFT

Iterative-FFT.Code.

0,4,2,6,1,5,3,7→000,100,010,110,001,101,011,111→000,001,010,011,100,101,110,111

```
BIT-REVERSE-COPY(a,A)
n←length [a]
for k←0 to n-1
      do A[rev(k)]←a_k
ITERATIVE-FFT
        BIT-REVERSE-COPY(a,A)
        n←length [a]
        for s←1 to log n
              do m←2^s
                  ω_m ←e^(2πi/m)
         for j←0 to n-1 by m ←1
                    for j←0 to m/2-1
                         do for  k←j to n-1 by m
                                   do t← ωA[k+m/2]
                                       u←A[k]
                                        A[k]←u+t
                                        A[k+m/2]←u-t
        ω← ω ω_m
        return A
```

A parallel FFT circuit



Problem: evaluating all derivatives of a polynomial at a point

a. Given coefficients $b_0, b_1, \ldots, b_{n-1}$ such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j$$

Show how to compute $A^{(t)}(x_0)$, for $t=0,1,2,\ldots,n-1$, in $O(n)$ time.

b. Explain how to find $b_0, b_1, \ldots, b_{n-1}$ in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k=0,1,2,\ldots,n-1$.

Problem: Toeplitz matrices

A **Toeplitz matrix** is an n × n matrix $A = (a_{ij})$, such that $a_{ij} = a_{i-1,j-1}$ for i=2,3,…,n and j=2,3,…,n.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b. Describe how to represent a Toeplitz matrix so that two n × n Toeplitz matrices can be added in O(n) time.

c. Give an O(n lg n)-time algorithm for multiplying an n × n Toeplitz matrix by a vector of length n. Use your representation from part (b).

d. Give an efficient algorithm for multiplying two n × n Toeplitz matrices. Analyze its running time.