Lecture 4: Polynomial Algorithms

Piotr Sankowski sankowski@dis.uniroma1.it
Stefano Leonardi leon@dis.uniroma1.it

Theoretical Computer Science

29.10.2009

Lecture Overview

- Introduction
- Point Polynomial Multiplication
- Fast Fourier Transform FFT
- Inverse Fourier Transform
- Polynomial Division
- Multipoint Polynomial Evaluation
- Polynomial Interpolation

Problems

The naive naive multiplication algorithm requires $\Theta(n^2)$ time.

We will show that using Fast Fourier Transform one can obtain algorithm that for all basic operations on polynommials working just slightly slower then $\Theta(n)$.

We will show how to :

- multiply polynomials in $O(n \log n)$ time,
- divide polynomials in $O(n \log n)$ time,
- compute interpolating polynomial in $O(n \log^2 n)$ time,
- compute n-values of in $O(n \log^3 n)$ time.

Point Multiplication

Let $A(x) = \sum_{i=0}^{n-1} a_i x^i$ and let $B(x) = \sum_{i=0}^{n-1} b_i x^i$ be two polynomials of degree n over field F.

The polynomials can be uniquely defined by their values in n different points.

Theorem 1 (Polynomial Interpolation) *For any set of n pairs $X = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ such that all values are x_i are pairwise different, there exists exactly one polynomial $C(x)$ of degree at most n such that $C(x_i) = y_i$ for $i = 0, 1, \ldots, n-1$.*

Point Multiplication

Let X be a fixed set of $2n$ points $x_0, \ldots, x_{2n-1} \in F$.

For this set we can define set of values for polynomials A and B:

$$X_A = \{(x_0, A(x_0)), (x_1, A(x_1), \ldots, (x_{2n-1}, A(x_{2n-1}))\},$$

$$X_B = \{(x_0, B(x_0)), (x_1, B(x_1), \ldots, (x_{2n-1}, B(x_{2n-1}))\}.$$

Let C be the product of A and B, then we have:

$$C(x_i) = A(x_i) \cdot B(x_i).$$

Point Multiplication

Because the degree of C is not higher then $2n$, so by Polynomial Interpolation theorem the set of values:

$$X_{A\cdot B}=\{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1), \ldots \\ \ldots, (x_{2n-1}, A(x_{2n-1})B(x_{2n-1}))\},$$

uniquely defines the polynomial $A \cdot B$.

Given the sets X_A and X_B we can determine X_C in $O(n)$ time. Next using it we can find the polynomial C.

Point Multiplication

The multiplication procedure is shown on the animation `ilustr_u.swf`.

Nevertheless in order to obtain faster algorithm than the naive one, we need to:

- compute n values of a polynomial faster then in $\Theta(n^2)$ operations,
- compute interpolating polynomial equally fast.

Fast Fourier Transform

In order to solve these both problems we will use Fast Furrier Transform (FFT).

In the above algorithm we did not assume anything about the set X.

The main idea in the construction of FFT is the choice of the right set X in order to guarentee that the many computations are repeated.

Fast Fourier Transform

Let us assume that we want to compute the value of the polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and that n is even.

If n is odd we add to $A(x)$ the monomial $0x^{n+1}$ what does not change the result.

We define the points $X_n = \{x_0, x_1, \ldots, x_{n-1}\}$ in the following way:

$$x_k = e^{\frac{2\pi k i}{n}}. \tag{1}$$

Fast Fourier Transform

For the polynomial $A(x)$ we define to new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ by choosing into them from $A(x)$ odd and even coefficients:

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

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

The polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ are of degree at most $\frac{n}{2}$.

Moreover we have:

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

Fast Fourier Transform

The evaluation of $A(x)$ in points $x_0, x_1, \ldots, x_{n-1}$ can be reduced to:

- evaluation of $A^{[0]}(x)$ and $A^{[1]}(x)$ for

$$X' = \{x_0^2, x_1^2, \ldots, x_{n-1}^2\}.$$

- computation of the values of $A(x)$ according to (2).

By the definition of x_i we have:

$$x_k^2 = \left(e^{\frac{2\pi ki}{n}}\right)^2 = e^{\frac{2\pi ki}{n/2}}.$$

Fast Fourier Transform

Note that we have

$$x_k^2 = x_{k+\frac{n}{2}}^2, \qquad so \qquad X' = X_{\frac{n}{2}}.$$

Hence, we reduced the problem of size n – computation of n values of degree n polynomial A

to two problems of size $\frac{n}{2}$ – computation of $\frac{n}{2}$ values of degree $\frac{n}{2}$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$.

Applying the recursion we get the FFT algorithm.

Fast Fourier Transform

- `if` n is odd `then` add $a_n = 0$ to A
- `if` $n = 1$ `then return` a_0
- $\omega_n = e^{\frac{2\pi i}{n}}$ and $\omega = 1$
- $a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$ and $a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
- $y^{[0]} = \mathrm{F}FT(a^{[0]})$ and $y^{[1]} = \mathrm{F}FT(a^{[1]})$
- `for` $k = 0$ `to` $\frac{n}{2} - 1$ `do`
  - $y_k = y_k^{[0]} + \omega y_k^{[1]}$
  - $y_{k+\frac{n}{2}} = y_k^{[0]} - \omega y_k^{[1]}$
  - $\omega = \omega\omega_n$
- `return` y

Fast Fourier Transform

The above algorithm starts by computing FFT of $A^{[0]}(x)$ and $A^{[1]}(x)$.

Next it combines the results to compute FFT of $A(x)$.

Let us look on the run of the algorithm.

Note that in the k-th step of the loop we have:

$$\omega = \omega_n^k = e^{\frac{2\pi ik}{n}} = x_k.$$

Fast Fourier Transform

Therefore

$$\begin{aligned} y_k = y_k^{[0]} + x_k y_k^{[1]} &= A^{[0]}(e^{\frac{2\pi ik}{n/2}}) + x_k A^{[1]}(e^{\frac{2\pi ik}{n/2}}) = \\ &= A^{[0]}\left(\left(e^{\frac{2\pi ik}{n}}\right)^2\right) + x_k A^{[1]}\left(\left(e^{\frac{2\pi ik}{n}}\right)^2\right) = \\ &= A^{[0]}(x_k^2) + x_k A^{[1]}(x_k^2) = A(x_k), \end{aligned}$$

and

$$\begin{aligned} y_{k+\frac{n}{2}} = y_k^{[0]} - x_k y_k^{[1]} &= A^{[0]}(e^{\frac{2\pi ik}{n/2}}) - e^{\frac{2\pi ik}{n}} A^{[1]}(e^{\frac{2\pi ik}{n/2}}) = \\ &= A^{[0]}(e^{\frac{2\pi i(k+n/2)}{n/2}}) + e^{\frac{2\pi ik+n/2}{n}} A^{[1]}(e^{\frac{2\pi i(k+n/2)}{n/2}}) = \\ &= A^{[0]}(x^2_{k+n/2}) + x^2_{k+n/2} A^{[1]}(x^2_{k+n/2}) = A(x_{k+\frac{n}{2}}). \end{aligned}$$

Fast Fourier Transform

In the last equality we have used (2).

This proves that the algorithm correctly computes FFT for $A(x)$.

The recursive equation for the running time of FFT is:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n).$$

and implies $T(n) = \Theta(n \log n)$.

Fast Fourier Transform

The animation `ilustr_a.swf` shows the run of the FFT algorithm for polynomial $A(x) = x^3 + 2x^2 + 5x + 3$.

The values of $A(x)$ are computed out of the values of $A^{[0]}(x)$ and $A^{[1]}(x)$, whereas the values of these polynomials are computed out of values of $A^{[00]}(x)$, $A^{[01]}(x)$, $A^{[10]}(x)$ and $A^{[11]}(x)$.

Inverse FFT

In order to finish the presentation of the fast polynomial multiplication algorithm we need to show how to compute the interpolating polynomial for the set of points X_n.

We will show that the computation of FFT can be understood as matrix-vector multiplication.

Next, we will show that FFT – its matrix, is almost its own inverse.

Inverse FFT

The FFT is a matrix-vector multiplication

$$(A(x_0), A(x_1), \ldots, A(x_{n-1}))^T = V_n(a_0, a_1, \ldots, a_{n-1})^T,$$

where $V_n = V(x_0, \ldots, x_{n-1})$ is a Vendermonde matrix:

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & x_0^3 & \cdots & x_0^{n-1} \\ x_1^0 & x_1^1 & x_1^2 & x_1^3 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & x_2^3 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & x_{n-1}^3 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Inverse FFT

The elements of the matrix $V(x_0, \ldots, x_{n-1})$ are given as

$$(V_n)_{j,k} = V(x_0, \ldots, x_{n-1})_{j,k} = x_j^k.$$

Using the definition of X_n we get:

$$V(x_0, \ldots, x_{n-1})_{j,k} = \left(e^{\frac{2\pi ij}{n}}\right)^k = e^{\frac{2\pi ijk}{n}}.$$

The inverse of FFT is represented as matrix-vector multipication

$$V_n^{-1}(A(x_0), A(x_1), \ldots, A(x_{n-1}))^T.$$

Inverse FFT

Lemma 1 *The matrix W_n defined as:*

$$(W_n)_{j,k} = \frac{1}{n} e^{\frac{-2\pi ijk}{n}},$$

is the inverse matrix for V_n.

We will show that

$$V_n W_n = I.$$

Let us consider an element (j,k) in the product matrix $V_n W_n$:

Inverse FFT

$$\begin{aligned}(V_n W_n)_{j,k} &= \sum_{l=0}^{n-1} (V_n)_{j,l} (W_n)_{l,k} = \\ &= \sum_{l=0}^{n-1} e^{\frac{2\pi i j l}{n}} \frac{1}{n} e^{\frac{-2\pi i l k}{n}} = \\ &= \sum_{l=0}^{n-1} \frac{1}{n} e^{\frac{2\pi l (j-k)}{n}} =\end{aligned}$$

If $j = k$ then $e^{\frac{2\pi k(j-k)}{n}} = 1$ and the sum equals 1.

Otherwise we get the sum of a geometric series:

$$= \frac{1}{n} \cdot \frac{1 - e^{\frac{2\pi n(j-k)}{n}}}{1 - e^{\frac{2\pi (j-k)}{n}}} = \frac{1}{n} \cdot \frac{1 - 1^{(j-k)}}{1 - e^{\frac{2\pi (j-k)}{n}}} = 0.$$

Hence $V_n W_n = I$.

Inverse FFT

The matrix V_n is defined as:

$$(V_n)_{j,k} = x_j^k = e^{\frac{2\pi ijk}{n}},$$

whereas the matrix W_n is defined as:

$$(W_n)_{j,k} = \frac{1}{n} e^{\frac{-2\pi ijk}{n}}.$$

Therefore in order to compute the inverse FFT we neeed to change $\omega_m = e^{\frac{2\pi i}{n}}$ to $\omega_m = e^{-\frac{2\pi i}{n}}$ in the algorithm and divide the result by n.

Polynomial Division

Let $A(x)$ be a polynomial of degree m and let $B(x)$ be the polynomial of degree n.

Without loss of generality we assume that $b_{n-1} \neq 0$.

In this problem we would like to compute two polynomials $D(x)$ and $R(x)$ such that:

$$A(x) = D(x)B(x) + R(x), \tag{3}$$

and degree of $R(x)$ is smaller then n.

We say that $D(x)$ is the *result* and $R(x)$ is the *remainder*.

Polynomial Division

The first idea is to compute the inverse of $B(x)$ and then use FFT to multiply $A(x)$ by it.

Unfortunately the polynomials have no inverses that would be polynomials.

We can extend our domain to guarantee existence of inverses.

Polynomial Division

We will make all computation in the set of formal series $F[[x]]$ over the field F.

Formal series are śimply"polynomials of infinite degree.

For some elements of $F[[x]]$ the inverses exist.

These elements have form $a + xA(x)$, where $a \neq 0$ and $A(x) \in F[[x]]$.

Polynomial Division

Question 1
What is the inverse of the series $1 + x$?

The inverse is equal to $\sum_{i=0}^{\infty}(-x)^i$.

Question 2
What is the inverse of the series $\sum_{j=0}^{\infty}(j+1)x^j$?

The inverse is equal to $(1-x)^2$.

Polynomial Division

Let us plug in $x = \frac{1}{z}$ into (3) to get:

$$A^R(z) = D^R(z)B^R(z) + z^{m-n-1}R^R(z) = \\ = B^R(z)D^R(z) \mod z^{m-n-1},$$

where $A^R(z) = z^m A(\frac{1}{z})$, $D^R(z) = z^{m-n} D(\frac{1}{z})$, $B^R(z) = z^n B(\frac{1}{z})$ and $R^R(z) = z^{n-1} R(\frac{1}{z})$, are polynomials with the reversed order of coefficients.

From the assumption that $b_{n-1} \neq 0$ we know that B^R is invertible.

Polynomial Division

Now we can write :

$$D^R(z) = A^R(x)\left(B^R(z)\right)^{-1} \mod z^{m-n-1}. \quad (4)$$

Note that in order to compute $D^R(z)$ we need only $m-n-1$ terms from series $\left(B^R(z)\right)^{-1}$.

The higher degree terms will disappear after the modulo operation by z^{m-n-1}.

Polynomial Division

The first t terms of the inverse of $A(x) = \sum_{i=0}^{n-1} a_i x^i, m)$ can be computed in the following way:

INVERSE$(A(x), t)$:

- `if` $t = 1$ `then return` $1/a_0$
- $A^{[0]}(x) = \text{INVERSE}(A(x), \lceil \frac{t}{2} \rceil)$
- `return`
  $\left(A^{[0]}(x) - \left[A(x)A^{[0]}(x) - 1\right] A^{[0]}(x)\right) \mod x^t.$

Polynomial Division

The computation is correct because:

$$\frac{1}{A(x)} - A^{[0]}(x)$$

is a multiple of $x^{\lceil \frac{t}{2} \rceil}$, so

$$\begin{aligned}\tfrac{1}{A(x)} - \left(A^{[0]}(x) - \left[A(x)A^{[0]}(x) - 1\right] A^{[0]}(x)\right) = \\ = A(x)\left(\tfrac{1}{A(x)} - A^{[0]}(x)\right)^2,\end{aligned}$$

is a multiple of x^t or x^{t+1}.

Polynomial Division

It we use fast polynomial multiplication to compute $(A^{[0]}(x) - (A(x)A^{[0]}(x) - 1)A^{[0]}(x))$ then the working time of the algorithm will be:

$$\begin{aligned}\sum_{i=0}^{\log n} O(2^i \log 2^i) &= O(1) \sum_{i=0}^{\log n} i2^i = \\ &= O(1) \cdot (2 + 2^{\log n+1}(\log n - 2)) = \\ &= O(n \log n).\end{aligned}$$

We are now ready to give an algorithm that divides $A(x)$ by $B(x)$ in $O(m \log m)$ time, where m is the degree of $A(x)$.

Polynomial Division

The algorithm dividing $A(x)$ by $B(x)$.

- $A^R(z) = z^m A(\frac{1}{z})$
- $B^R(z) = z^n B(\frac{1}{z})$
- $(B^R(z)^{-1})(z) = \text{INVERSE}(B^R(z), m-n-1)$
- $D^R(z) = A^R(x) \left(B^R(z)\right)^{-1} \mod z^{m-n-1}$
- $D(x) = z^{m-n-1} D^R(\frac{1}{x})$
- $R(x) = A(x) - D(x)B(x)$
- `return` $(D(x), R(x))$

Multi-point Evaluation

In this problem we are given a degree n polynomial $A(x)$ and a set of n points $x_1, \ldots, x_n$ and need to compute n values $A(x_1), \ldots, A(x_n)$.

We will show how to reduce the problem to two smaller problems.

In order to do this we will use polynomial division.

Multi-point Evaluation

Let us define for $A(x)$ two new polynomials:

$$A^{[0]}(x) = A(x) \mod \prod_{i=0}^{\frac{n}{2}-1}(x - x_i),$$

$$A^{[1]}(x) = A(x) \mod \prod_{i=\frac{n}{2}}^{n-1}(x - x_i).$$

from properties of division we have:

$$A(x_i) = A^{[0]}(x_i) \quad \text{for} \quad i = 0, \ldots, \frac{n}{2} - 1$$

$$A(x_i) = A^{[1]}(x_i) \quad \text{for} \quad i = \frac{n}{2}, \ldots, n - 1$$

Multi-point Evaluation

Polynomials $A^{[0]}(x)$ and $A^{[0]}(x)$ are of degree $\frac{n}{2}$.

Using polynomial division we can compute them in $O(n \log n)$ time.

This recurrence gives working time

$$T(n) = 2T(\frac{n}{2}) + \Theta(n \log n).$$

and implies $T(n) = \Theta(n \log^2 n)$.

Interpolation

For the given set of pairs $X = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ such that all x_i are pairwise different we would like to find interpolating polynomial. wielomian interpolacyjny.

We will show how to reduce the problem to two smaller problems.

In order to do this we will use multi-point polynomial evaluation.

Let is define $P(x)$ as: $P(x) = \prod_{i=0}^{\frac{n}{2}-1}(x - x_i)$.

Interpolation

We start by recursively computing a polynomial $A^{[0]}(x)$ of degree $\frac{n}{2}$ interpolating the set $X^{[0]} = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{\frac{n}{2}-1}, y_{\frac{n}{2}-1})\}$.

Now the polynomial $A(x)$ can be written as:

$$A(x) = A^{[0]}(x) + P(x)A^{[1]}(x),$$

where $A^{[1]}$ is an unknown polynomial of degree $\frac{n}{2}$.

By the construction $A(x)$ is an interpolating polynomial for $X^{[0]}$.

Interpolation

Note that we need to solve an interpolation problem to find $A^{[1]}$, because:

$$A^{[1]}(x_i) = \frac{y_i - A^{[0]}(x_i)}{P(x_i)} \quad \text{for} \quad i = \frac{n}{2}, \ldots, n-1.$$

We only need to compute:

- the values of $A^{[0]}(x)$ for the set of points $\{x_{\frac{n}{2}}, \ldots, x_{n-1}\}$.
- the values of $P(x)$ for the set of points $\{x_{\frac{n}{2}}, \ldots, x_{n-1}\}$.

We have reduced the problem to two smaller ones. This reduction give an algorithm working in $O(n \log^3 n)$.