FAST POLYNOMIAL MULTIPLICATION, DIVISION, EVALUATION & INTERPOLATION

LUIS FERRONI

Abstract. We make a brief survey on several applications of the Fast Fourier Transform on problems relating operations with polynomials. These notes were written as part of the lectures given for the Argentinian ICPC Training Camp held virtually during the COVID-19 global pandemic. The exposition is based on [1] and [2].

1. Introduction

Let us start with two polynomials A and B with real coefficients, as follows:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m,$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n,$$

where we have that $a_m \neq 0$ and $b_n \neq 0$. We say that the *degree* of A and B are respectively m and n and write:

$$\deg(A) = m \text{ and } \deg(B) = n.$$

We first consider the problem of computing the product of these two polynomials, $P(x) \doteq A(x)B(x)$. Performing the multiplication directly we of course obtain that:

$$P(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{m+n} x^{m+n},$$

where for each $0 \leq k \leq m + n$ we have:

$$p_k = \sum_{j=0}^{k} a_j b_{k-j}.$$

The expression on the right is what we call a *convolution*. It is evident that if j is bigger than m, the value of a_j is not defined. In that case we assume that $a_j = 0$, and this does not alter our computations. Of course, we do the same for b_{k-j} for $k - j > n$.

To compute the product $P(x)$ in this way (coefficient by coefficient), we make it coefficient by coefficient:

$$\begin{aligned}
&p_0 = a_0 b_0 && \longrightarrow \text{ 1 product and no additions,}\\
&p_1 = a_0 b_1 + a_1 b_0 && \longrightarrow \text{ 2 products and 1 addition,}\\
&p_2 = a_0 b_2 + a_1 b_1 + a_2 b_0 && \longrightarrow \text{ 3 products and 2 additions,}\\
&\quad\vdots
\end{aligned}$$

Notice that every product $a_i b_j$ for each $0 \leq i \leq m$ and $0 \leq j \leq n$ appears exactly once in all these computations (exactly in the term x^{i+j} in $P(x)$). So, we have in total $(m + 1)(n + 1) = O(mn)$ products and $O(mn)$ additions.

What we want to do is to show a faster way, in the sense of performing less operations, to compute $P(x)$.

2. Fast Multiplication

Let us recall some facts about n-th roots of unity. First of all, to make ideas more concrete let us consider the case of cubic roots of 1. This is, all $z \in \mathbb{C}$ such that:

$$z^3 = 1.$$

Of course, $z = 1$ is a solution, but it is the *less interesting*. However, notice that the equation $z^3 = 1$ translates into $z^3 - 1 = 0$ which, in turn, is equivalent to:

$$(z-1)(z^2 + z + 1) = 0.$$

So, what if we consider the quadratic $z^2 + z + 1 = 0$? We get two complex solutions:

$$z_1 = -\frac{1}{2} + i\frac{\sqrt{3}}{2},$$

$$z_2 = -\frac{1}{2} - i\frac{\sqrt{3}}{2}.$$

Can we find easy expressions for the solutions of $z^n = 1$ for n bigger than 3? The answer is affirmative.

Theorem 2.1 (De Moivre)**.** *Let n be a positive integer. Then there are exactly n complex numbers satisfying $z^n = 1$ and they are given by:*

$$\begin{aligned}
z_0 &= 1 \\
z_1 &= \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right) \\
z_2 &= \cos\left(2\cdot\frac{2\pi}{n}\right) + i\sin\left(2\cdot\frac{2\pi}{n}\right) \\
&\vdots \\
z_{n-1} &= \cos\left((n-1)\frac{2\pi}{n}\right) + i\sin\left((n-1)\frac{2\pi}{n}\right)
\end{aligned}$$

Proof. Any complex-analysis or algebra book. □

What is important here is the following identity (which can also be proven with standard methods). For each $0 \leq k \leq n-1$.

$$z_1^k = z_k.$$

Besides, notice that from the identity:

$$z^n - 1 = (z-1)(z^{n-1} + z^{n-2} + \ldots + z + 1),$$

we have that all z_i for $i = 1, \ldots, n-1$ (i.e. all of those $z_i \neq 1$), we have that:

$$z_i^{n-1} + z_i^{n-2} + \ldots + z_i + 1 = 0.$$

We are going to put a name to our root z_1, and to its inverse, which happens to be very special.

Definition 2.2. Let n be a positive integer. We will denote:

$$\omega_n \doteq \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right).$$

And we call ω_n an n-th *primitive root of unity*. Also we denote:

$$\xi_n \doteq \cos\left((n-1)\frac{2\pi}{n}\right) + i\sin\left((n-1)\frac{2\pi}{n}\right).$$

The word *primitive* comes from the fact that if we elevate ω_n to all the numbers $0, 1, \ldots, n-1$ we end up obtaining *all* roots of unity. There are many n-th primitive roots (exactly $\varphi(n)$ to be precise), but we won't address that here.

Notice that as we said before:

$$\omega_n^{n-1} = \xi_n,$$

and since $\omega_n^n = 1$ (because it is an n-th root of unity) we have that:

$$\xi_n = \frac{1}{\omega_n}.$$

Definition 2.3. Let n be a positive integer. We define the *n-th Fourier Matrix* $\mathbf{F}_n$ as the following $n \times n$ matrix of complex numbers:

$$\mathbf{F}_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \xi_n & \xi_n^2 & \cdots & \xi_n^{n-1} \\ 1 & \xi_n^2 & \xi_n^4 & \cdots & \xi_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_n^{n-1} & \xi_n^{2(n-1)} & \cdots & \xi_n^{(n-1)(n-1)} \end{bmatrix}$$

This is, the entry (i, j) (indexing from zero) consists of the element ξ_n^{ij}.

Example 2.4. For example the matrices $\mathbf{F}_2$ and $\mathbf{F}_3$ are given by:

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$\mathbf{F}_3 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -\frac{1}{2} - i\frac{\sqrt{3}}{2} & -\frac{1}{2} + i\frac{\sqrt{3}}{2} \\ 1 & -\frac{1}{2} + i\frac{\sqrt{3}}{2} & -\frac{1}{2} - i\frac{\sqrt{3}}{2} \end{bmatrix}.$$

Definition 2.5. Let $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ be a vector of complex numbers. We define the *Discrete Fourier Transform* of $\mathbf{x}$ to be the vector obtained by multiplicating $\mathbf{x}$ by $\mathbf{F}_n$.

$$\mathbf{y} = \mathbf{F}_n\mathbf{x}.$$

Since $\mathbf{F}_n$ is just a matrix, it is natural to ask if it has an inverse. The answer is yes, and it is *almost the same matrix*!

Theorem 2.6. *Let n be a positive integer, the inverse of the Fourier Matrix $\mathbf{F}_n$ is given by:*

$$\mathbf{F}_n^{-1} = \frac{1}{n}\overline{\mathbf{F}_n}.$$

Where the bar indicates that every element inside the matrix has to be conjugated in the sense that $\overline{a + ib} \doteq a - ib$.

Proof. You have to perform the multiplication $\mathbf{F}_n \cdot \overline{\mathbf{F}_n}$ and verify using properties of the roots of unity that the result is nI where I is the identity matrix. □

So, this says that calculating the *inverse transform* is more or less the same as transforming itself. Concretely:

$$\mathbf{F}_n^{-1}\mathbf{x} = \frac{1}{n}\overline{\mathbf{F}_n\overline{x}}.$$

Now, all these relates nicely with the product of polynomials. First of all, if we have two vectors of the same size:

$$\mathbf{x} = (x_0, x_1, \ldots, x_{m-1}),$$

$$\mathbf{y} = (y_0, y_1, \ldots, y_{m-1}),$$

we will denote by $\mathbf{x} \odot \mathbf{y}$ the vector with the products coordinate by coordinate. It is:

$$\mathbf{x} \odot \mathbf{y} = (x_0y_0, x_1y_1, \ldots, x_{m-1}y_{m-1}).$$

The following theorem is the one relating everything.

Theorem 2.7. *Let A and B be polynomials*

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{m-1}x^{m-1},$$
$$B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1},$$

such that $m \geq n$. Let $C(x) = A(x)B(x)$ be the product of the polynomials, with coefficients given by:

$$C(x) = c_0 + c_1x + \ldots + c_{m+n-2}x^{m+n-2},$$

Consider the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ of $2m$ coordinates given by:

$$\mathbf{a} = (a_0, a_1, \ldots, a_{m-1}, \underbrace{0, 0, \ldots, 0}_{m})$$
$$\mathbf{b} = (b_0, b_1, \ldots, b_{n-1}, \underbrace{0, 0, \ldots, 0}_{2m-n})$$
$$\mathbf{c} = (c_0, c_1, \ldots, c_{m+n-2}, \underbrace{0, \ldots, 0}_{m-n+1})$$

Then:

$$\mathbf{F}_{2m}\mathbf{c} = (\mathbf{F}_{2m}\mathbf{a}) \odot (\mathbf{F}_{2m}\mathbf{b}).$$

Proof. Several calculations. □

So, to put it mildly, if you want to compute all the coefficients of a product of polynomials, then perform the following steps:

(1) First transform $A(x)$ and $B(x)$ (completing to $2m$ elements).
(2) Then make the product of these two new vectors coordinate by coordinate.
(3) Then anti-transform.

So we know that the product between a matrix and a vector can be done in $O(m^2)$ if the matrix has size $m \times m$. However for a Fourier matrix we can do much better. We will see that it is possible to perform the product $\mathbf{F}_m\mathbf{x}$ in $O(m\log(m))$.

So step (1) above can be done in $O((2m)\log(2m)) = O(m\log(m))$. Step (2) can be done in $O(2m)$ trivially. And step (3) can be done in $O((2m)\log(2m))$ thanks to the fact that anti-transforming is essentially just transforming and conjugating.

Theorem 2.8. *Let $n = 2^r$. Then it holds:*

$$\mathbf{F}_n = \begin{bmatrix} \mathbf{F}_{n/2} & \mathbf{D}_{n/2}\mathbf{F}_{n/2} \\ \mathbf{F}_{n/2} & -\mathbf{D}_{n/2}\mathbf{F}_{n/2} \end{bmatrix} \mathbf{P}_n$$

where $\mathbf{D}_{n/2}$ is a diagonal matrix and $\mathbf{P}_n$ is a permutation matrix given by:

$$\mathbf{D}_{n/2} = \begin{bmatrix} 1 & & & & \\ & \xi_n & & & \\ & & \xi_n^2 & & \\ & & & \ddots & \\ & & & & \xi_n^{n/2-1} \end{bmatrix} \quad \textit{and} \quad \mathbf{P}_n^T = \begin{bmatrix} | & | & | & & | & | & | & & | \\ e_0 & e_2 & e_4 & \dots & e_{n-2} & e_1 & e_3 & \dots & e_{n-1} \\ | & | & | & & | & | & | & & | \end{bmatrix}$$

Proof. Do the computations. □

Now, the precedent theorem implies that performing one Fourier transform of order $n = 2^r$ is equivalent to applying two transforms of order $n/2 = 2^{r-1}$.

input : A vector $\mathbf{x} = (x_0, \dots, x_{n-1})$ where $n = 2^r$
output: The Fourier Transform of $\mathbf{x}$, namely $\mathbf{F}_n\mathbf{x}$
Function `FourierTransform`($\mathbf{x}$):
  if $n = 1$ **then**
    return $\mathbf{x}$;
  end
  $\mathbf{x}_0 := (x_0, x_2, \dots, x_{n-2})$;
  $\mathbf{x}_1 := (x_1, x_3, \dots, x_{n-1})$;
  $\mathbf{y}_0 :=$ `FourierTransform`($\mathbf{x}_0$);
  $\mathbf{y}_1 :=$ `FourierTransform`($\mathbf{x}_1$);
  for $i \in \{0, 1, \dots, n/2 - 1\}$ **do**
    $\mathbf{y}_{1i} := \mathbf{y}_{1i} \cdot \xi_n^i$
  end
  $\mathbf{a} := \mathbf{y}_0 + \mathbf{y}_1$;
  $\mathbf{b} := \mathbf{y}_0 - \mathbf{y}_1$;
  return $(a_0, a_1, \dots, a_{n/2-1}, b_0, b_1, \dots, b_{n/2-1})$;

So now we state the main result:

Theorem 2.9. *Let A and B be polynomials*

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{m-1}x^{m-1},$$
$$B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1},$$

such that $m \geq n$. The following algorithm computes the coefficients $(c_0, c_1, \dots, c_{m+n-2})$ of the product

$$C(x) \doteq A(x)B(x) = \sum_{j=0}^{m+n-2} c_i x^i.$$

Moreover, the overall complexity is $O(m \log(m))$.

input : *Two polynomials A and B as above of degrees m and n, $m \geq n$*
output: *The product of both polynomials $A(x)B(x)$*
Function `multiply`(A, B)*:*
$M := 2^{\lceil \log_2(2m) \rceil}$ *(first power of 2 exceeding $2m$);*
$\mathbf{a} := (a_0, a_1, \ldots, a_{m-1}, \underbrace{0, 0, \ldots, 0}_{M-m})$*;*
$\mathbf{b} := (b_0, b_1, \ldots, b_{n-1}, \underbrace{0, 0, \ldots, 0}_{M-n})$*;*
$\widetilde{\mathbf{a}} :=$ `FourierTransform`$(\mathbf{a})$*;*
$\widetilde{\mathbf{b}} :=$ `FourierTransform`$(\mathbf{b})$*;*
$\widetilde{\mathbf{c}} := \widetilde{\mathbf{a}} \odot \widetilde{\mathbf{b}}$*;*
$\widetilde{\mathbf{c}} :=$ `ComplexConjugate`$(\widetilde{\mathbf{c}})$*;*
$\mathbf{c} :=$ `FourierTransform`$(\widetilde{\mathbf{c}})$*;*
$\mathbf{c} = \frac{1}{M} \cdot$ `ComplexConjugate`$(\mathbf{c})$*;*
return $c_0 + c_1 x + \ldots + c_{m+n-2} x^{m+n-2}$*;*

3. Fast Division

Now that we know fast algorithms to perform polynomial multiplication, we can try to use them to find a fast way of dividing two polynomials.

Theorem 3.1 (Polynomial Division)**.** *Let $A(x)$ and $B(x)$ two polynomials with real coefficients such that $\deg(A) = m$ and $\deg(B) = n$ with $m \geq n$. Then there exist polynomials Q and R such that $\deg(Q) = m - n$, $\deg(R) < n$ and:*

$$A(x) = Q(x)B(x) + R(x).$$

Furthermore, they are unique.

We will develop an algorithm to compute Q and R with complexity $O(m \log(m))$. This is much better than the usual $O(mn)$ that one gets by performing the usual high-school algorithm for polynomial division.

Let us start with a remark about an interpretation of what is to *invert* a polynomial. First of all, in the realm of *formal series*, the following is a customary (and *famous*) identity:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \ldots.$$

The insight is that there's nothing special with the polynomial $1 - x$. It is just the case that the series we obtain on the right is particularly easy for that polynomial, but we can find analogous expressions for *almost all polynomials.*

Theorem 3.2. *Let B be a polynomial with real coefficients:*

$$B(x) = b_0 + b_1 x + \ldots + b_n x^n.$$

Such that $b_0 \neq 0$. Then there exists an infinite sequence of real values $q_0, q_1, q_2, \ldots$ such that:

$$\frac{1}{B(x)} = \sum_{j=0}^{\infty} q_j x^j,$$

for all x in a sufficiently small neighborhood of the real number 0.

Remark 3.3. The condition of $b_0 \neq 0$ only means that $B(0) \neq 0$.

So Theorem 3.2 tells us that for some polynomials we can find an expression for its inverse, but what we obtain is a *infinite polynomial* (a series). If we call

$$C(x) \doteq \sum_{j=0}^{\infty} q_j x^j,$$

we of course have that:

$$B(x)C(x) = 1,$$

So, considering C just as an infinite polynomial we may just look at the coefficient of x^k for $k = 0, 1, 2, \ldots$, in the left hand side. We see that:

$$b_0 q_0 = 1,$$

$$\sum_{j=0}^{k} b_j q_{k-j} = 0 \text{ for all } k = 1, 2, 3, \ldots$$

In particular, we obtain a formula for q_0 as simply $\frac{1}{b_0}$ and then we can compute recursively all values of q_j for $j \geq 1$. For example, we have that:

$$b_0 q_1 + b_1 q_0 = 0,$$

From where we obtain:

$$b_0 q_1 + \frac{b_1}{b_0} = 0,$$

And that is $q_1 = \frac{-b_1}{b_0^2}$. Going on this way one can compute all values for each j. Notice that if $B(x)$ has integer coefficients, in general $C(x) = \frac{1}{B(x)}$ has *rational* coefficients.

Theorem 3.4. *Let $B(x) = b_0 + b_1 x + \ldots + b_n x^n$ be a polynomial such that $b_0, b_n \neq 0$. The following pseudocode provides an algorithm to compute the first $d+1$ terms of the formal series $\frac{1}{B(x)}$ with complexity $O((n+d)\log(n)\log(d))$.*

input : $B = b_0 + b_1 x + \ldots + b_n x^n$ *with* $b_0, b_n \neq 0$ *and an integer* $d \geq 0$
output: *First* $d+1$ *terms of* $\frac{1}{B(x)}$*, it is:* $q_0 + q_1 x + \ldots + q_d x^d$
Function `invert`*(B,d):*
  if $d = 0$ **then**
    return $\frac{1}{b_0}$*;*
  end
  $C(x) :=$ `invert`$(B, \lfloor \frac{d}{2} \rfloor)$*;*
  $Q(x) := C(x)(2 - C(x)B(x))$*;*
  return `truncate`$(Q(x), d)$*;*

Where the line $Q(x) = C(x)(2 - C(x)B(x))$ is executed using the FFT algorithm and the instruction `truncate`$(Q(x), d)$ *truncates the polynomial $Q(x)$ up to degree d.*

Proof. To prove the correctness of the algorithm, note that if we set $C(x) =$ `invert`$(B, \lfloor \frac{d}{2} \rfloor)$, we have that the formal series: $\frac{1}{B(x)} - C(x)$ is a multiple of $x^{\lfloor \frac{d}{2} \rfloor + 1}$.

In particular, notice that we have the equality:

$$\frac{1}{B(x)} - C(x)(2 - B(x)C(x)) = \frac{1}{B(x)} - 2C(x) + B(x)C(x)^2$$

$$= B(x)\left(\frac{1}{B(x)^2} - 2C(x)\frac{1}{B(x)} + C(x)^2\right)$$
$$= B(x)\left(\frac{1}{B(x)} - C(x)\right)^2$$

And since we noticed that $\frac{1}{B(x)} - C(x)$ is a multiple of $x^{\lfloor \frac{d}{2} \rfloor + 1}$, this means that the $\frac{1}{B(x)} - C(x)(2 - B(x)C(x))$ is for sure a multiple of x^{d+1}. Hence $\frac{1}{B(x)}$ and $C(x)(2 - B(x)C(x))$ coincide up to degree d, as desired. □

Now let us solve the problem of dividing two polynomials. Let us consider

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m,$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n,$$

where $m \geq n$. We know that there are two polynomials Q and R such that:

$$Q(x) = q_0 + q_1 x + \ldots + q_{m-n} x^{m-n},$$

and that $\deg(R) < n$ and such that the following equality holds:

$$A(x) = Q(x)B(x) + R(x).$$

We may assume safely that $R(x) = r_0 + r_1 x + \ldots + r_{n-1} x^{n-1}$ where we allow all r_j to be zero.

Notice that evaluating the previous equality in $\frac{1}{x}$ we get:

$$A\left(\tfrac{1}{x}\right) = Q\left(\tfrac{1}{x}\right) B\left(\tfrac{1}{x}\right) + R\left(\tfrac{1}{x}\right).$$

But we have that:

$$A\left(\tfrac{1}{x}\right) = \frac{1}{x^m}\left(a_m + a_{m-1}x + \ldots + a_1 x^{m-1} + a_0 x^m\right)$$
$$B\left(\tfrac{1}{x}\right) = \frac{1}{x^n}\left(b_n + a_{b-1}x + \ldots + b_1 x^{n-1} + b_0 x^n\right)$$
$$Q\left(\tfrac{1}{x}\right) = \frac{1}{x^{m-n}}\left(a_{m-n} + a_{m-n-1}x + \ldots + a_1 x^{m-n-1} + a_0 x^{m-n}\right)$$

In particular, if we use the notation $\widetilde{F}$ to denote the *reversed* expression for a polynomial F, we see that:

$$\frac{1}{x^m}\widetilde{A}(x) = \frac{1}{x^{m-n}}\widetilde{Q}(x)\frac{1}{x^n}\widetilde{B}(x) + \frac{1}{x^{\deg R}}\widetilde{R}(x),$$

which implies:

$$\widetilde{A}(x) = \widetilde{Q}(x)\widetilde{B}(x) + x^{m-\deg(R)}\widetilde{R}(x).$$

This last equality implies that $\widetilde{A}(x)$ and $\widetilde{Q}(x)\widetilde{B}(x)$ have the same coefficients up to degree $m - \deg(R)$, and since $\deg(R) < n$, this ensures that they coincide up to degree $m - n + 1$.

All amounts to say that

$$\widetilde{A}(x) \cdot \frac{1}{\widetilde{B}(x)} = \widetilde{Q}(x) + x^{m-\deg(R)}\widetilde{R}(x) \cdot \frac{1}{\widetilde{B}(x)}$$

So, this all amounts to prove the correctness of the following algorithm to compute $Q(x)$ and, of course, $R(x) = A(x) - Q(x)B(x)$.

Theorem 3.5. *Let $A(x)$ and $B(x)$ be polynomials of degree m and n. The following pseudocode provides an algorithm to compute the quotient $Q(x)$ and $R(x)$ with complexity $O(m\log(m))$.*

input : *$A(x)$ and $B(x)$ polynomials of degree m and n*
output: *$\{Q, R\}$ the quotient and remainder of dividing A by B*
Function `divide`*(A,B):*
  if $\deg B > \deg A$ **then**
    return $\{0, A\}$;
  end
  $\widetilde{A} :=$ `revert`(A);
  $\widetilde{B} :=$ `revert`(B);
  $\widetilde{Q} := \widetilde{A} \cdot$ `invert`$(\widetilde{B}, m-n)$;
  $Q :=$ `revert`$(\widetilde{Q}) \cdot x^{m-n-\deg(Q)}$;
  $R := A(x)B(x) - Q(x)$;
  return $\{Q, R\}$;

Where the instruction `revert` *does revert the coefficients of a polynomial and all products are computed using the FFT algorithm.*

4. Fast Multi-Evaluation

Now we approach the problem of evaluating a polynomial $A(x) = a_0 + a_1x + \ldots + a_mx^m$ of degree m into n real numbers: $x_1, \ldots, x_n$.

In other words, we want to compute $A(x_1), \ldots, A(x_n)$ in a fast way.

Notice that evaluating the polynomial into just one point x_1 is essentially the same thing as dividing $A(x)$ into $x - x_1$. More explicitly, if we perform that division, we get:

$$A(x) = Q(x)(x - x_1) + R(x),$$

where $R(x)$ is actually a polynomial of degree < 1, it is, a *constant*. Say $R(x) = r_0$ Notice that putting $x = x_1$ in both sides, we get:

$$A(x_1) = Q(x_1)(x_1 - x_1) + R(x_1),$$

which translates into:

$$A(x_1) = r_0.$$

And we have computed $A(x_1)$, as we wanted.

Of course performing this division to compute $A(x_1)$ is probably not the best way to do such evaluation for just one point. However, if we have many points this idea helps.

For evaluating $A(x)$ into $x_1, \ldots, x_n$ we form these two polynomials:

$$F(x) = (x - x_1) \cdot \ldots \cdot (x - x_{\lfloor \frac{n}{2} \rfloor}),$$
$$G(x) = (x - x_{\lfloor \frac{n}{2} \rfloor + 1}) \cdot \ldots \cdot (x - x_n).$$

Now let's divide $A(x)$ by $F(x)$ and $G(x)$ to obtain:

$$A(x) = Q_1(x)F(x) + R_1(x),$$

$$A(x) = Q_2(x)G(x) + R_2(x).$$

Where we have that $\deg(R_1) < \deg(F) = \lfloor \frac{n}{2} \rfloor$ and that $\deg(R_2) < \deg(G) = \lceil \frac{n}{2} \rceil$.

The fact that $F(x_i) = 0$ for $i = 1, \ldots, \lfloor \frac{n}{2} \rfloor$ guarantees that $A(x_i) = R(x_i)$ for those points. So now we reduced the problem to evaluate a polynomial of degree less than $\lfloor \frac{n}{2} \rfloor$ into $\lfloor \frac{n}{2} \rfloor$ points.

The same happens with the indices $i = \lfloor \frac{n}{2} \rfloor + 1, \ldots, n$ and $R_2(x)$.

So, to be short, we have that:

$$A(x_i) = \begin{cases} R_1(x_i) & \text{if } i = 1, \ldots, \lfloor \frac{n}{2} \rfloor, \\ R_2(x_i) & \text{otherwise.} \end{cases}$$

Now taking into account that we know how to multiply polynomials fastly to calculate $F(x)$ and $G(x)$ and we know how to divide A by them to obtain $R_1(x)$ and $R_2(x)$, we can iterate this process. This proves the following result.

Theorem 4.1. *Let $A(x)$ be a polynomial of degree m, and consider $x_1, \ldots, x_n$ given. The following pseudocode provides an algorithm to compute $A(x_1), \ldots, A(x_n)$ with complexity $O((m+n)\log(n)(\log(n) + \log(m)))$.*

input : $A(x)$ *of degree* m *and a* n*-uple* $\vec{x} = (x_1, \ldots, x_n)$
output: $(A(x_1), A(x_2), \ldots, A(x_n))$
Function `multi_evaluation`$(A, \vec{x})$*:*
  $F(x) := (x - x_1) \cdot \ldots \cdot (x - x_{\lfloor \frac{n}{2} \rfloor})$;
  $G(x) := (x - x_{\lfloor \frac{n}{2} \rfloor + 1}) \cdot \ldots \cdot (x - x_n)$;
  $(Q_1, R_1) :=$ `divide`(A, F);
  $(Q_2, R_2) :=$ `divide`(A, G);
  $L_1 :=$ `multi_evaluation` $\Big(R_1, (x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor})\Big)$;
  $L_2 :=$ `multi_evaluation` $\Big(R_2, (x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n)\Big)$;
  return `concatenate`(L_1, L_2);

Where the instruction `concatenate`(L_1, L_2) *concatenates the vectors L_1 and L_2 in that order and the polynomials F and G were already processed (see Remark below).*

Remark 4.2. It is important to remark that to avoid an extra $\log(n)$ factor on the complexity one should *preprocess* all polynomials F and G as appearing inside the algorithm. Otherwise one can compute in each call of `multi_evaluation` F and G in $O(n\log^2(n))$ using a divide and conquer recursion and FFT. In that case the overall complexity of the above code is $O((m+n)\log^2(n)\log(m))$ but it is possible to implement it much more easily.

This preprocessing as follows: for simplicity assume that n is a power of 2 (if it is not, then complete the vector $\vec{x}$ of values with zeros until you arrive to a power of 2, this will not affect the overall complexity at all).

You can turn the whole algorithm to an *iterative* version of multi-evaluation, by using a binary tree.

If $n = 2^k$ consider the following.

input : A n-uple $\vec{x} = (x_1, \ldots, x_n)$, where $n = 2^k$
output: An array Tree of $2n$ elements such that Tree$[i]$ stores the product of all $(x - x_j)$ for j such that $j + 2^{k-1}$ is a descendant of j
Function `FillTree`$(\vec{x})$:
  for $i \in \{2^k + 1, \ldots, 2^{k+1}\}$ **do**
    Tree$[i] := x - x_{i-2^k}$;
  end
  for $i \in \{2^k, 2^k - 1, \ldots, 2\}$ **do**
    Tree$[i] :=$ Tree$[2i - 1] \cdot$ Tree$[2i]$;
  end
  return Tree;

Of course the second loop is executed with decreasing values of i and using the FFT for the polynomial multiplication. As an example, we end up getting that:

$$\text{Tree}[2] = (x - x_1) \cdot \ldots \cdot (x - x_n).$$
$$\text{Tree}[3] = (x - x_1) \cdot \ldots \cdot (x - x_{\frac{n}{2}}).$$
$$\text{Tree}[4] = (x - x_{\frac{n}{2}+1}) \cdot \ldots \cdot (x - x_n).$$

And this whole preprocess may be performed in $O(n \log^2(n))$ time as one can verify easily.

5. Fast Interpolation

If one has two points (x_1, y_1) and (x_2, y_2) in the plane, it is clear that there is a unique *line* (it is, a polynomial of degree 1) passing through both of them. This can be generalized:

Theorem 5.1. *Let $(x_1, y_1), \ldots, (x_n, y_n)$ be a set of points in $\mathbb{R}^2$ such that $x_i \neq x_j$ for all $i \neq j$. There exists a unique polynomial $A(x)$ of degree at most $n-1$ that satisfies that:*

$$A(x_1) = y_1,$$
$$A(x_2) = y_2,$$
$$\vdots$$
$$A(x_n) = y_n.$$

Furthermore, it is possible to find an expression for this unique polynomial $A(x)$. Let us consider the polynomial p defined by:

$$p(x) \doteq (x - x_1)(x - x_2) \cdot \ldots \cdot (x - x_n).$$

And now, for each $1 \leq i \leq n$, the polynomial:

$$p_i(x) = \frac{p(x)}{x - x_i} = (x - x_1) \cdot \ldots \cdot (x - x_{i-1})(x - x_{i+1}) \cdot \ldots \cdot (x - x_n)$$

It is clear that $\deg(p) = n$ and that $\deg(p_i) = n - 1$ for each $1 \leq i \leq n$.

Notice that p_i has a special property: $p_i(x_j) = 0$ for all $j \neq i$. In particular, if we call $d_i = p_i(x_i)$ it is evident that $d_i \neq 0$. So now we have that the polynomial given by:

$$\frac{1}{d_i} p_i(x).$$

Evaluates to 0 for $x = x_j$ for $j \neq i$ and evaluates to 1 for $x = x_i$. This allows us to make a smart *linear combination* to obtain our polynomial $A(x)$. Namely:

$$A(x) = \sum_{i=1}^{n} y_i \frac{1}{d_i} p_i(x).$$

Notice that evaluating in $x = x_j$ on the right cancels out for all indices except $i = j$ for which it gives $y_j \frac{1}{d_j} p_j(x_j) = y_j$, as desired.

Proposition 5.2. *For all $1 \leq i \leq n$ it holds that:*

$$p'(x_i) = d_i$$

Proof. We have that:

$$p(x) = (x - x_i)p_i(x).$$

So, using the product rule:

$$p'(x) = (x - x_i)p_i'(x) + p_i(x).$$

Now, evaluating on both sides gives $p'(x_i) = p_i(x_i) = d_i$, as desired. □

So now, notice that we can get the derivative of p in linear time using that the derivative of x^j is jx^{j-1} and summing. Also we can evaluate this derivative in the points $(x_1, \ldots, x_n)$ in $O(n \log^2(n))$ time using the algorithm of the precedent section.

So we have the fractions $\frac{y_i}{d_i}$ in total $O(n \log^2(n))$ time. Now to perform the whole summation and obtain $p(x)$ we can do as follows.

Again, as in the previous section we assume that $n = 2^k$. An important disclaimer is that here we cannot add random points (x_i, y_i) to complete the list up to a power of two, because that would change the interpolating polynomial. See the remark below.

input : A n-uple of points $(x_1, y_1), \ldots, (x_n, y_n)$, where $n = 2^k$ and $x_i \neq x_j$
output: An array IntTree of $2n$ elements such that IntTree[i] stores a polynomial interpolating all (x_j, y_j) for j a descendant of i
Function `FillInterpolationTree`$(x_1, y_1, \ldots, x_n, y_n)$:
    `FillTree`$(x_1, \ldots, x_n)$;
    $p(x) :=$ Tree[2];
    $p'(x) :=$ `derivative`(p);
    $(d_1, \ldots, d_n) :=$ `multi_evaluation`$(p'(x), (x_1, \ldots, x_n))$;
    for $i \in \{2^k + 1, \ldots, 2^{k+1}\}$ **do**
        IntTree[i] $:= \dfrac{y_{i-2^k}}{d_{i-2^k}}$;
    end
    for $i \in \{2^k, 2^k - 1, \ldots, 2\}$ **do**
        IntTree[i] := Tree[$2i - 1$] $\cdot$ IntTree[$2i$] + Tree[$2i$] $\cdot$ IntTree[$2i - 1$];
    end
    return IntTree;

We can see that if we perform the products using the FFT, the overall complexity of executing the algorithm above is $O(n \log^2(n))$.

Moreover, the following is the important thing:

Theorem 5.3. *The value stored in* IntTree[2] *is the polynomial of degree at most* $n-1$ *that interpolates* $(x_1, y_1), \ldots, (x_n, y_n)$*. Hence we have an algorithm to compute the interpolating polynomial of* n *points in* $O(n \log^2(n))$*.*

Remark 5.4. Before we prove this result, let us have a word on the case on which n is not a power of 2. It is possible to show that if in the above algorithm in the places where there is a 2^k one simply puts an n, it will still work perfectly (of course, one should do the same in the case of the algorithm `FillTree`)). This resolves the problem of interpolation in $O(n \log^2(n))$ for arbitrary n.

Proof. We proceed by induction. We claim that

$$\text{IntTree}[i] = \sum_{j \leftarrow i} \frac{y_j}{d_j} \frac{\text{Tree}[i]}{x - x_j}.$$

Where the notation $j \leftarrow i$ means that $j + 2^k$ is a leaf descendant of i. Observe that since $\text{Tree}[2] = (x-x_1)\cdot\ldots\cdot(x-x_n)$ it will be evident from our formula for Lagrange's polynomial that the result follows.

Now, we verify our base case for $i = 2^k + 1, \ldots, 2^{k+1}$, we do have that $\text{Tree}[i] = x - x_{i-2^k}$ and it is its only leaf descendant, so we get $\text{IntTree}[i] = \frac{y_j}{d_j}$ and it is the case.

Now, suppose that it is true for $2i$ and $2i-1$. Then:

$$\begin{aligned}\text{IntTree}[i] &= \text{Tree}[2i-1] \cdot \text{IntTree}[2i] + \text{Tree}[2i] \cdot \text{IntTree}[2i-1] \\ &= \text{Tree}[2i-1] \sum_{j \leftarrow 2i} \frac{y_j}{d_j} \frac{\text{Tree}[2i]}{x - x_j} + \text{Tree}[2i] \sum_{j \leftarrow 2i-1} \frac{y_j}{d_j} \frac{\text{Tree}[2i-1]}{x - x_j}\end{aligned}$$

And since all the leafs descending from $2i-1$ and all the leafs descending from $2i$ form a disjoint set and they are precisely the leafs descending from i we get that:

$$\text{IntTree}[i] = \sum_{j \leftarrow i} \frac{y_j}{d_j} \frac{\text{Tree}[i]}{x - x_j}.$$

And the induction follows. □

References

[1] A. V. Aho and J. E. Hopcroft, *The Design and Analysis of Computer Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1974.

[2] C. D. Meyer, *Matrix Analysis and Applied Linear Algebra*. USA: Society for Industrial and Applied Mathematics, 2000.

Università di Bologna, Dipartimento di Matematica, Piazza di Porta San Donato, 5, 40126 Bologna BO - Italia

Email address: ferroniluis@gmail.com