Divide and Conquer: Polynomial Multiplication via FFT[1]

In the last lecture, we saw how two polynomials can be multiplied fast by using divide-and-conquer. In this supplement we cover a different approach to multiplying polynomials, but divide-and-conquer will still be the strategy which would give the improvement. There are many pieces to this algorithm, and let us first conceptually develop these pieces.

Polynomial Multiplication via Evaluation and Interpolation

In the polynomial multiplication problem, we are given *coefficients* of two polynomials $P(x)$ and $Q(x)$, and we have to output the *coefficients* of the product $R(x) = P(x) \cdot Q(x)$. Suppose we had a different problem. Suppose that all we have to do is to construct an "oracle" which takes input a number a, and outputs $R(a)$. Thus, this oracle would be an *implicit representation* of the product polynomial $R(x)$. In how much time can you implement this oracle?

This is easy. We evaluate $P(a)$. We evaluate $Q(a)$. We multiply $P(a) \cdot Q(a)$ and return the answer. Assuming number additions and multiplications are $O(1)$ time operations, we see that the running time is driven by the *polynomial evaluation* time for calculating $P(a)$ and $Q(a)$. Both of these take $O(n)$ time, and thus, the implicit representation of R takes $O(n)$ time *per query*.

How does this help us come closer to the problem of finding the *coefficients* of R? Well, here is the main insight which one recalls from ones algebra classes in high-school[2]. Since R is a degree $2n$ polynomial, if we knew the values of R at $2n + 1$ *distinct* numbers $a_0, a_1, \ldots, a_{2n}$, then we can obtain the coefficients of R. This step is called *interpolation*. Indeed, think of the case when R is a degree 1 polynomial (a linear function). If we knew $A := R_0 + R_1 a$ and $B := R_0 + R_1 b$ for two distinct numbers a and b, then we can figure R_0 and R_1 out by a solving a system of simultaneous equations[3].

So the main schema for polynomial multiplication is the following three steps:

a. Select $2n + 1$ distinct numbers $a_0, a_1, \ldots, a_{2n}$.

b. Evaluate $R(a_0), R(a_1), \ldots, R(a_{2n})$.

c. Interpolate.

However, as you can see there are big issues. The naive way of evaluating $R(a)$ takes $O(n)$ time. Therefore, Step 2 seems to take $O(n^2)$ time, which is a show-stopper in itself (and we haven't even gone to Step 3). The second observation, and the key observation, is the following : we have some freedom in choosing the distinct numbers $a_0, a_1, \ldots, a_{2n}$. Is it possible to choose them in a clever way such that all the $R(a_i)$'s can be together figured out in a much faster way? The answer is affirmative, but to get there we first shift our viewpoint.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

[2] I guess only the case of degree-1 polynomials is covered in high-school?

[3] Indeed, $R_1 = \frac{A-B}{a-b}$ and $R_0 = A - aR_1 = \frac{aB-Ab}{a-b}$.

Polynomial Evaluation and Interpolation via Matrix-Vector Multiplication

Let $P(x) = P_0 + P_1 x + \cdots + P_n x^n$ be a degree n polynomial. Given a number a, one think of the evaluation of $P(a)$ as a vector-vector multiplication as follows.

$$P(a) = \begin{pmatrix} 1 & a & a^2 & \cdots & a^n \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ \vdots \\ P_n \end{pmatrix}$$

Remark: *This way of looking a polynomial evaluation is a very useful viewpoint helpful in many different settings.*

And therefore, if we have to evaluate the polynomial P at $n+1$ different locations, then the values at all these locations can be expressed as a *matrix-vector* multiplication as follows.

$$\begin{pmatrix} P(a_0) \\ P(a_1) \\ P(a_2) \\ \vdots \\ P(a_n) \end{pmatrix} = \begin{pmatrix} 1 & a_0 & a_0^2 & \cdots & a_0^n \\ 1 & a_1 & a_1^2 & \cdots & a_1^n \\ 1 & a_2 & a_2^2 & \cdots & a_2^n \\ \vdots & & \ddots & & \vdots \\ 1 & a_n & a_n^2 & \cdots & a_n^n \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ \vdots \\ P_n \end{pmatrix}$$

In plain English this says that the evaluation of the polynomial at $n+1$ locations is the product of a very special looking matrix with the vector of coefficients. The matrix is special in that on every row, the entries are the different powers of some numbers. These matrices have a name; they are called *Vandermonde matrices*.

One very important property of Vandermonde matrices is that if all the a_i's are distinct, then the matrix is *invertible*. This property is basically the interpolation property. For note that if M^{-1} is the inverse of the matrix M above, then if we knew the value of $P(a_i)$ for $0 \le i \le n$, then the coefficients $P_0, P_1, \ldots, P_n$ can be found by multiplying M^{-1} with the vector of the $P(a_i)$'s. If you are interested, you should check this for the case of $n = 1$ and see that it gives the solution to the two simultaneous equations.

To get back to our algorithm, we would now like to find $a_0, a_1, \ldots, a_n$ such that (a) the matrix-vector product $M \cdot v$, where M is the resulting Vandermonde matrix and v is the vector of coefficients of P and Q, can be found much fast (for evaluation), (b) the inverse M^{-1} can be found fast, and finally,(c) $M^{-1} \cdot v$, where v is not the evaluation of R at different points, can also be found fast (for interpolation). "Fast" here means much better than $O(n^2)$ time. If we could find such magical a_i's, then indeed we would have found a fast algorithm for multiplying polynomials. To find them, we need to venture into the complex plane.

Complex Roots of Unity

Since we need to find a_i's such that the resulting Vandermonde matrix M can be multiplied with a vector v fast, one way would be to select a_i's such that the rows look similar. Of course, if we picked all a_i's the same, then we would get this for free. But is there something else we can pick?

A quick refresher on complex numbers. In real numbers, 1 has only one cube-root: $+1$. But, if one looks at the equation $z^3 = 1$, then this is a degree 3 equation, and it in fact has 3 "roots" or solutions. One is

$z = 1$. The others happen to be complex numbers. Indeed, they are $\omega = e^{2i\pi/3}$ and $\omega^2 = e^{4i\pi/3}$. In general, there are n-roots of unity for any number n, and they are $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2i\pi/n}$. Note that $\omega^n = e^{2i\pi} = 1$ by Euler's identity. Also note if n is even, then $\omega^{n/2} = -1$.

Now, for our problem at hand, let $(a_0, a_1, \ldots, a_n)$ be the $(n+1)$th complex roots of unity. It will be useful for us to assume $(n+1)$ is a power of 2. So, $n = 2^\ell - 1$. That is the $(n+1)$th roots of unity are are $(1, \omega, \omega^2, \ldots, \omega^{2^\ell - 1})$ where $\omega = e^{\frac{2i\pi}{2^\ell}}$. How does the Vandermonde matrix look like? Well, we see that it looks like

$$M_n(\omega) \;=\; \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^n \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^n & \omega^{2n} & \cdots & \omega \end{pmatrix}$$

Example of the case $\ell = 1, 2$ and 3 are as follows.

$$M_1(\omega) = \begin{pmatrix} 1 & 1 \\ 1 & \omega \end{pmatrix} \quad M_3(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} \quad M_7(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix}$$

In particular, if we index the rows from $j = 0$ to $2^\ell - 1$, and the columns also from $k = 0$ to $2^\ell - 1$, then the (j,k)th entry is precisely ω^{jk}. In particular, the matrix is symmetric. It has another beautiful property.

Claim 1. The inverse of the Vandermonde matrix $M_n(\omega)$ is the Vandermonde matrix $\frac{1}{n+1} \cdot M_n(\omega^n)$, where ω is the $(n+1)$th root of unity.

Proof. If we take the product of $M_n(\omega) \cdot M_n(\omega^n)$, and consider the (i,j)th entry, we get that it is

$$\sum_{k=0}^{n} M_n(\omega)[i,k] \cdot M_n(\omega^n)[k,j] \;=\; \sum_{k=0}^{n} \omega^{ik} \cdot (\omega^n)^{jk} \;=\; \sum_{k=0}^{n} \omega^{k\cdot(i+nj)}$$

Now, if $i = j$, then $i + nj = (n+1)i$, and thus each entry in the sum above is $\omega^{k\cdot(i+nj)} = \omega^{ki(n+1)} = 1$ since ω is the $(n+1)$th root of unity. Thus, if $i = j$, then the (i,j)th entry of $M_n(\omega) \cdot M_n(\omega^n)$ is $(n+1)$.

On the other hand if $i \neq j$, then $\hat{\omega} := \omega^{(i+nj)}$ is another $(n+1)$th root of unity. Thus, the sum above is a geometric sum of the roots of unity. This equals zero since it equals $\frac{1-\hat{\omega}^{(n+1)}}{1-\hat{\omega}}$, and the numerator is 0. Thus, $M_n(\omega) \cdot M_n(\omega^n)$ is a diagonal matrix with $(n+1)$ on the diagonal. This implies the claim. $\square$

Thus, if we can figure out a fast algorithm to multiply $M_n(\omega) \cdot v$ for some vector v when ω is a root of unity, then we also get for free the product $M_n(\omega)^{-1} \cdot v$. Next, we see the recursive structure of $M_n(\omega)$ which allows one to compute $M_n(\omega) \cdot v$ is $O(n \log n)$ time.

The Recursive Structure of $M_n(\omega)$

The way to look at this is to first reorder the matrix such that we first have the "even" columns of M and then the "odd" columns of M. Let me actually explain this using an example. Let $n = 3$, and then, we get (recall $\omega^4 = 1$) that

$$M_3(\omega) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix}$$

For concreteness, let's fix a vector $\vec{P} := (P_0, P_1, P_2, P_3)$, and we want to compute $M_3(\omega) \cdot \vec{P}$. So, we want to compute

$$\begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

When we shift the evens and odds, we also shift the W's and the P's to get that

$$\begin{pmatrix} W_0 \\ W_2 \\ W_1 \\ W_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega & \omega^3 \\ 1 & 1 & \omega^2 & \omega^2 \\ 1 & \omega^2 & \omega^3 & \omega \end{pmatrix} \cdot \begin{pmatrix} P_0 \\ P_2 \\ P_1 \\ P_3 \end{pmatrix}$$

Divide the matrix above into four 2×2 matrices $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$. And so, we get

$$\begin{pmatrix} W_0 \\ W_2 \end{pmatrix} = A \cdot \begin{pmatrix} P_0 \\ P_2 \end{pmatrix} + B \cdot \begin{pmatrix} P_1 \\ P_3 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} W_1 \\ W_3 \end{pmatrix} = C \cdot \begin{pmatrix} P_0 \\ P_2 \end{pmatrix} + D \cdot \begin{pmatrix} P_1 \\ P_3 \end{pmatrix}$$

So, we have replaced the matrix-vector product to 4 "smaller" matrix-vector products (plus some additions). As we know, this won't be enough for divide-and-conquer; this won't give anything better than $O(n^2)$. But now, we make our main observations:

(a) A itself is a 2×2 Vandermonde matrix corresponding to ω^2.

(b) The matrix $C = A$.

(c) The matrix B is a "shift" of A, in that, the first row of B is the same as first row of A, but the second row of B is ω times the second row of A.

(d) The matrix D is also a "shift" of A, in that, the first row of B is the same as first row of A, but the second row of B is $\omega^3 = -\omega$ times the second row of A.

Why is this useful? Well, this tells us that if we know $A \cdot \begin{pmatrix} P_0 \\ P_2 \end{pmatrix}$ and $A \cdot \begin{pmatrix} P_1 \\ P_3 \end{pmatrix}$, then we can calculate (W_0, W_1, W_2, W_3). Indeed, we can obtain $B \cdot \begin{pmatrix} P_1 \\ P_3 \end{pmatrix}$ by multiplying the second row of $A \cdot \begin{pmatrix} P_1 \\ P_3 \end{pmatrix}$ with ω. And adding this with $A \cdot \begin{pmatrix} P_0 \\ P_2 \end{pmatrix}$ gives $\begin{pmatrix} W_0 \\ W_2 \end{pmatrix}$. Similarly, we can obtain $\begin{pmatrix} W_1 \\ W_3 \end{pmatrix}$. Thus, we have reduced the problem to *two* matrix-vector products where the matrix is a Vandermonde matrix of half the dimension, and this is what leads to the super-saving.

Let us now give more details about the general case. First, by just padding the polynomial with 0's we may assume n is such that $n+1$ is a power of 2. So, $(n+1) = 2^\ell$. Let ω be an 2^ℓth root of unity. We consider the matrix $M_n(\omega)$ with $(n+1)$ rows and $(n+1)$ columns whose (j,k)th entry is ω^{jk}. We first shift the columns considering first the even columns and then the odd columns. Let this shifted matrix be called N. Given a vector v, we can construct $M_n(\omega)\cdot v$ by evaluating $N\cdot w$ where w is a shift of v by taking all even coordinates of v followed by odd-coordinates of v. We index the columns of N as $(0,2,4,\ldots,2^\ell-2)$ followed by $(1,3,5,\ldots,2^\ell-1)$. Now, we exploit the recursive substructure of N. We break $N = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, where A,B,C,D are four $2^{\ell-1}\times 2^{\ell-1}$ matrices.

$$\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\
1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\
1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\
1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\
1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\
1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\
1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega
\end{pmatrix}
\longrightarrow
\begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega^2 & \omega^4 & \omega^6 & \omega & \omega^3 & \omega^5 & \omega^7 \\
1 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\
1 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega & \omega^7 & \omega^5 \\
1 & 1 & 1 & 1 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\
1 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega & \omega^3 \\
1 & \omega^4 & 1 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\
1 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega
\end{pmatrix}$$

What's the (j,k)th entry of A? Note that this corresponds to the $(j,2k)$th entry of $M_n(\omega)$. And this would therefore be $\omega^{2jk} = (\omega^2)^{jk}$. What is the (j,k)th entry of C? This corresponds to the $(j+2^{\ell-1}, 2k)$th entry of $M_n(\omega)$. Therefore, this value is $\omega^{(j+2^{\ell-1})\cdot 2k} = \omega^{2jk}\cdot\omega^{2^\ell k} = (\omega^2)^{jk}$. The last equality follows because $\omega^{2^\ell} = 1$. Therefore, $A = C$.

How about B and D? What's the (j,k)th entry of B? This corresponds to the $(j, 2k+1)$th entry of $M_n(\omega)$. Therefore, this would be $\omega^{j(2k+1)} = \omega^j\cdot\left(\omega^2\right)^{jk}$. And finally, the (j,k)th entry of D is the $(j+2^{\ell-1}, 2k+1)$th entry of $M_n(\omega)$. Therefore, this value would be $\omega^{(j+2^{\ell-1})(2k+1)} = \omega^{2jk}\cdot\omega^{2^{\ell-1}}\cdot\omega^j\cdot\omega^{2^\ell k}$. Using the fact that $\omega^{2^{\ell-1}} = -1$, we get that this is $-\omega^j\cdot(\omega^2)^{jk}$. Thus, we get that (a) A is a Vandermonde matrix with $2^{\ell-1}$ rows and columns corresponding a root of unity of $2^{\ell-1}$, (b) $C = A$, (c) B is a "shifted" version of A where the jth row has been multiplied by ω^j, and (d) D is a shifted version of A where the jth row has been multiplied by $-\omega^j$. Below is another illustration

Therefore, to evaluate $M_{2^\ell}(\omega)\cdot v$, we first *recursively* evaluate $b_1 = M_{2^{\ell-1}}(\omega^2)\cdot v_1$ and $b_2 = M_{2^{\ell-1}}(\omega^2)\cdot v_2$, where v_1 is the vector formed by taking even coordinates of v and v_2 is the one obtained by taking odd coordinates of v. Next, we evaluate the shifted version of b_2 by multiplying the jth row with ω^j to obtain b_3. And then, we return the concatenation of (b_1+b_3) and (b_1-b_3).

```
1: procedure EvalPoly($\ell, v, \omega$): ▷ $v$ is an $2^\ell$-dimensional array; $\omega$ is a $2^\ell$th root of unity.
2:     ▷ Evaluate $M_{2^\ell-1}(\omega) \cdot v$. If $v$ contains the coefficients of a polynomial, then the matrix-vector product contains the evaluation of the polynomial at the $2^\ell$th roots of unity.
3:     ▷ Addition, Multiplication of complex numbers assumed to be $O(1)$ time.
4:     if $\ell = 0$ then:
5:         return $v$. ▷ Singleton Array
6:     $v_1$ be the $2^{\ell-1}$ dimensional vector comprising the even components of $v$.
7:     $v_2$ be the $2^{\ell-1}$ dimensional vector comprising the odd components of $v$.
8:     $b_1 \leftarrow$ EvalPoly($\ell - 1, v_1, \omega^2$).
9:     $b_2 \leftarrow$ EvalPoly($\ell - 1, v_2, \omega^2$).
10:    Compute $b_3$ from $b_2$ by multiplying $j$th coordinate by $\omega^j$, for $0 \leq j \leq 2^\ell - 1$.
11:    $w \leftarrow (b_1 + b_3) \circ (b_1 - b_3)$, where $\circ$ is the concatenation operator.
12:    Obtain $a$ from $w$ by "reshuffling": $a_0 = w_0$, $a_1 = w_{2^{\ell-1}+1}$, $a_2 = w_1$, $a_3 = w_{2^{\ell-1}+2}$, and so on.
13:    return $a$.
```

The running time of the above algorithm is governed by the recurrence $T(2^\ell) = 2T(2^{\ell-1}) + O(2^\ell)$ which equates to $O(\ell 2^\ell)$.

Theorem 1. Assuming complex numbers can be added and multiplied in $O(1)$ time, for any 2^ℓ dimensional vector v, EvalPoly(ℓ, v, ω) returns the vector $M_{2^\ell}(\omega) \cdot v$ in $O(n \log n)$ time, where $n = 2^\ell$.

Remark: *We should remark here that we are multiplying complex numbers in Line 10 and adding them in line Line 11. We should also mention that assuming addition and multiplication of complex numbers in $O(1)$ time is a* big *assumption. Indeed, this is a big assumption even for* real *numbers as the precision (how many bits after the decimal) to which we store it matters. Recall that the input to the problem has only integer coefficients. Indeed,* integer *polynomial multiplication needs to take care of this. And only very recently (in 2019) was an $O(n \log n)$ time algorithm for multiplying polynomials was announced, which assumed only $O(1)$ operations for integer arithmetic operations.*

The $O(n \log n)$ time polynomial multiplication algorithm is as follows.

```
1: procedure FFT-PolyMult($P, Q$): ▷ $P$ and $Q$ are degree $n$ polynomials
2:     Pick the smallest power of 2 which is bigger than $2n$. Let this be $2^\ell$.
3:     Consider $P$ and $Q$ as degree $2^\ell$ polynomials by substituting 0 whenever the degree $> n$.
4:     Abuse notation and let the arrays $P[0 : 2^\ell - 1]$ and $Q[0 : 2^\ell - 1]$ contain these coefficients.
5:     Let $\omega$ be a $2^\ell$th root of unity.
6:     $A_1 \leftarrow$ EvalPoly($\ell, P, \omega$). ▷ $A_1$ is an $2^\ell$-length array.
7:     $A_2 \leftarrow$ EvalPoly($\ell, Q, \omega$). ▷ $A_1$ is an $2^\ell$-length array.
8:     Obtain $A$ by coordinate-wise multiplying $A_1$ and $A_2$. ▷ $A$ is an $2^\ell$-length array.
9:     $R \leftarrow$ EvalPoly($\ell, A, \omega^{2^\ell-1}$)
10:    return $R$.
```