CS345 : Design and Analysis of Algorithms Spring 2014

Lecture 4: Multiplication of two Polynomials - II

We shall first do a quick recap of the last lecture.
Notation: Let ω_m denote the principle mth root of unity. Let S_m be the set of mth roots of unity. So $S_m = \{1, \omega_m, \omega_m^2, \cdots, \omega_m^{m-1}\}$. The following is the key observation for the case when m is even (mentioned in the last lecture).

> If we square the elements of S_m, we get $S_{m/2}$.

This key observation forms the basis for $O(n \log n)$ time for the Evaluation step of our 3-step algorithm for polynomial multiplication. Let **Evaluate**(A,n) be the algorithm which has two parameters. The first parameter is a polynomial of degree less than n expressed in coefficient representation; you may store them as an array or list of size n. The second parameter is n. It returns an array $V[0..2n-1]$ storing the value of the polynomial on $2n$th roots of unity. So the output is $\{A(\omega_{2n}^j) | 0 \leq j \leq 2n-1)\}$. The following is the pseudocode of **Evaluate**(A,n) with some blanks left.

Algorithm 1: Evaluate(A,n)

```
1  if n=1 then
2  |  ...;
3  |  ...;
4  else
5  |  Let A_even be the polynomial consisting of even coefficients of A;
6  |  Let A_odd be the polynomial consisting of odd coefficients of A;
7  |  V_even ← Evaluate(A_even, n/2);
8  |  V_odd ← Evaluate(A_odd, n/2);
9  |  foreach 0 ≤ j ≤ 2n − 1 do
10 |  |  V[j] ← V_even[ω_{2n}^{2j}] + ω_{2n}^j · V_odd[ω_{2n}^{2j}];
11 |  end
12 |  return V;
13 end
```

Homework: Attempt the following exercises:

1. Fill in the blanks left for the base case $n = 1$
2. Think of some suitable implementation (you may use some more variables) so that each iteration of the for loop takes $O(1)$ time.
3. Let $T(n, 2n)$ denote the time complexity of evaluating a polynomial of degree less than n on $2n$ complex roots of unity using **Evaluate**(A,n). Show that the following is the recurrence for $T(n, 2n)$:
$$T(n, 2n) = cn + 2T(n/2, n)$$
Show that this recurrence has the solution $T(n) = O(n \log n)$.

4. Provide the most efficient implementation of the algorithm given above. This is to test your programming skills.

5. Give an iterative implementation of the above algorithm.

Remark: We designed **Evaluate**(A,n) to evaluate an algorithm of degree less than n at $2n$th roots of unity. It is easy to make it uniform, that is, the number of points of evaluation and degree bound can be same. After all, a polynomial of degree less than n is also a polynomial of degree less than $2n$. Henceforth, we shall use **Evaluate**(A,n) to be an algorithm for evaluating a polynomial of degree less than n at nth roots of unity. Notice that n is still power of 2.

Interpolation

Given the $(point, value)$ representation of polynomial $C(x)$, the aim is to compute its coefficient representation. In particular, we are given the values $\{C(\omega_{2n}^j | 0 \leq j \leq 2n-1\}$ and we wish to compute the coefficients c_t of the polynomial $C(x)$. We gave a hint in the last class that this step will also be similar to the first step, that is, evaluation of some polynomial. Make an intelligent guess for the following question.

Question: What would be the polynomial which we should evaluate to compute c_t's ?

Given the $2n$ values $\{C(\omega_{2n}^j) | 0 \leq j \leq 2n-1\}$, it is natural to guess the polynomial as

$$D(x) = \sum_{\ell=0}^{2n-1} d_\ell x^\ell, \quad \text{where} \quad d_\ell = C(\omega_{2n}^\ell) \tag{1}$$

It leads to the following natural question ?

Question: What should we evaluate $D(x)$ on ?

Obviously, since we want this task to take $O(n \log n)$ time, we must evaluate $D(x)$ on $2n$th complex roots of unity. It leads to the following natural question.

Question: How to compute or extract the coefficients of $C(x)$ once we have $\{D(\omega_{2n}^k) | 0 \leq k \leq 2n-1)\}$?

For this purpose, we should look closely at what is $D(\omega_{2n}^k)$.

$$\begin{aligned} D(\omega_{2n}^k) &= \sum_{\ell=0}^{2n-1} d_\ell \cdot (\omega_{2n}^k)^\ell \\ &= \sum_{\ell=0}^{2n-1} C(\omega_{2n}^\ell) \cdot (\omega_{2n}^k)^\ell \qquad \text{using Equation 1} \\ &= \sum_{\ell=0}^{2n-1} \left(\sum_{j=0}^{2n-1} c_j \cdot (\omega_{2n}^\ell)^j \cdot (\omega_{2n}^k)^\ell \right) \end{aligned}$$

This is double summation. To simplify it, try to answer the following question.

Question: If you expand the last equation above, what will be the expression containing the term c_j ?

It will be $\sum_{\ell=0}^{2n-1} \cdot(\omega_{2n}^{\ell})^j \cdot (\omega_{2n}^{k})^{\ell} \cdot c_j$; hence we can rearrange the above equation above to get

$$\begin{aligned} D(\omega_{2n}^{k}) &= \sum_{j=0}^{2n-1} \left(c_j \cdot \sum_{\ell=0}^{2n-1} (\omega_{2n}^{\ell})^j \cdot (\omega_{2n}^{k})^{\ell} \right) \\ &= \sum_{j=0}^{2n-1} \left(c_j \cdot \sum_{\ell=0}^{2n-1} \omega_{2n}^{(j+k)\ell} \right) \end{aligned}$$

What would be $\sum_{\ell=0}^{2n-1} \omega_{2n}^{(j+k)\ell}$? Clearly $\omega_{2n}^{(j+k)}$ is one of the $2n$th complex roots of unity. So this expression is summation of powers of this root. To see what it is finally, we shall again use our elementary knowledge of complex roots of unity. There are two cases now.

Case 1: $j + k$ is multiple of $2n$.
In this case, $\omega_{2n}^{(j+k)} = 1$, so the value of the expression is $2n$.

Case 2: $j + k$ is not multiple of $2n$.
So $\omega_{2n}^{(j+k)} \neq 1$. Let $q = j + k$. So using summation formula of geometric series,

$$\sum_{\ell=0}^{2n-1} \omega_{2n}^{q\ell} = \frac{\omega_{2n}^{2nq} - 1}{\omega_{2n}^{q} - 1} = \frac{(\omega_{2n}^{2n})^q - 1}{\omega_{2n}^{q} - 1} = \frac{0}{\neq 0} = 0$$

Hence based on these two cases, we can observe that $D(\omega_{2n}^{k}) = 2n \cdot c_{2n-k}$. In other words, coefficients $c_t, 0 \leq 2n-1$, of polynomial $C(x)$ are related to $D(x)$ as follows.

$$c_t = \frac{D(\omega_{2n}^{2n-t})}{2n}$$

So here is the interpolation algorithm. The input is the polynomial $D(x) = \sum_{\ell=0} d_\ell x^\ell$ where $d_\ell = C(\omega_{2n}^{\ell})$. The output is an array C such that $C[t] = c_t$

Algorithm 2: Interpolate(D)

```
1 V ←Evaluate(D, 2n) ;
2 foreach 0 ≤ t ≤ 2n − 1 do
3 |   C[t] ← V[2n−t]/2n;
4 end
5 return C;
```

It follows that the running time of the algorithm is $O(n \log n)$. This concludes the last step of our three step algorithm for multiplying two polynomials of degree less than n. The total complexity of these three steps is $O(n \log n)$.

Here are some homework problems for you.

Homework Exercise :

1. Since $A(x)$ and $B(x)$ are polynomials of degree less than n, so $C(x) = A(x) \cdot B(x)$ is a polynomial of degree less than $2n-1$. But, during interpolation, we compute $2n$ coefficients. Does it cause any problem ? Can we say anything about (the value of) some coefficient of $C(x)$ computed during interpolation ?

2. Recall the algorithm of transforming $A(x)$ into (point,value) representation. For this purpose, we expressed $A(x)$ as a function of $A_{even}(x)$ and $A_{odd}(x)$. We used the fact $A_{even}(x)$ and $A_{odd}(x)$ are of degree less than $n/2$. There is alternate way to express $A(x)$ as a function of two polynimials of degree less than $n/2$ as follows. $A_{left}(x) = \sum_{j=0}^{(n-1)/2-1} a_j x^j$, $A_{right}(x) = \sum_{j=0}^{(n-1)/2} a_{j+(n-1)/2} x^j$. Can we use this representation to design $O(n \log n)$ time algorithm for this task ? Where does it fail ?

3. If you wish to test yourself whether you have understood the fundamentals of divide and conquer technique, do the exercises of practice sheet 1.

4. You will get a programming assignment which will involve an efficient implementation of this algorithm. Please make sincere and honest attempts to design an efficient implementation of this novel algorithm. It will help you in long run.