NORTHWESTERN UNIVERSITY

EECS 336 – Homework 3

Weihan Chu

April 26, 2016

1 QUESTION 1

The FFT method uses a divide and conquer strategy. The normal FFT using the even-index and odd-index coefficients of A(x). We now define the five new degree-bound n/5 polynomials $A^{[0]}(x), A^{[1]}(x), A^{[2]}(x), A^{[3]}(x), A^{[4]}(x)$

$$A^{[0]}(x) = a_0 + a_5 x + a_{10} x^2 + ... + a_{n-5} x^{n/5-1}$$
$$A^{[1]}(x) = a_1 + a_6 x + a_{11} x^2 + ... + a_{n-4} x^{n/5-1}$$
$$A^{[2]}(x) = a_2 + a_7 x + a_{12} x^2 + ... + a_{n-3} x^{n/5-1}$$
$$A^{[3]}(x) = a_3 + a_8 x + a_{13} x^2 + ... + a_{n-2} x^{n/5-1}$$
$$A^{[4]}(x) = a_4 + a_9 x + a_{14} x^2 + ... + a_{n-1} x^{n/5-1}$$

$$A(X) = A^{[0]}(x^5) + xA^{[1]}(x^5) + x^2 A^{[2]}(x^5) + x^3 A^{[3]}(x^5) + x^4 A^{[4]}(x^5)$$

so that problem of evaluation A(x) at $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ reduces to:
1. Evaluating the degree-bound n/5 polynomials $A^{[0]}(x), A^{[1]}(x), A^{[2]}(x), A^{[3]}(x), A^{[4]}(x)$ at the points $(\omega_n^0)^5, (\omega_n^1)^5, ..., (\omega_n^{n-1})^5$
2.combining the results according to equation
From the lemma we have, the values twice. Therefore, the polynomials $A^{[0]}(x), A^{[1]}(x), A^{[2]}(x)$, $A^{[3]}(x), A^{[4]}(x)$ are recursively evaluated and the sub-problems have same form as the original problem, but half the size. We now can divide the DFT_n into two $DFT_{N/3}$ computations. So we can use FFT algorithm to computes the DFT of an n-element vector $a = (a_0, a_1, ..., a_{n-1})$, where n is a power of 5.
So we can achieve the whole process by three steps below:

1. Compute point value representations of A(x), B(x) of length 5n through the application of FFT. This step takes $\Theta(nlgn)$
2. Compute the point-value representation for the equation $C(x) = A(x)B(x)$, this step takes$\Theta(n)$
3. Compute coefficient representation of the polynomial C(x) through the FFT on 5n point value to compute the inverse DFT, this step takes $\Theta(nlgn)$
So the total running time is $\Theta(nlgn)$

2 QUESTION 2

In order to find the Cartesian sum of A,B and C. we need to represent A,B and C in polynomial form. So they can be represented as

$$A(x) = x^{a_0} + x^{a_1} + x^{a_2} ..... + x^{a_(n-1)} + x^{a_n}$$

$$B(x) = x^{b_0} + x^{b_1} + x^{b_2} ..... + x^{b_(n-1)} + x^{b_n}$$

$$C(x) = x^{c_0} + x^{c_1} + x^{c_2} ..... + x^{c_(n-1)} + x^{c_n}$$

As the maximum range of sets A, B and C are in $10n$, so the polynomial $A(x), B(x), C(X)$ are of degrees at most 10n.
We first compute A(x) and B(x), then their result with C(x) to get final result.
As the first step, the sum of two polynomials of degree-bound 10n is a polynomial of degree-bound 20n. So we first need to add 10n high-order coefficients of 0. Then we need to compute (20n)th roots of unity. We use the FFT to compute the polynomial product of A(x) and B(x) in $O(nlgn)$ times with follow algorithm
Then the sum of two polynomials of degree-bound 10n and 20n is a polynomial of degree-bound 30n, so we first need to add 10n high-order coefficients of 0 to the product of A(x) and B(x), then add 20n high-order coefficients of 0 to C(x). Now we use the 30n-th roots of unity to computer the final result in $O(nlgn)$ times with follow algorithm
And the algorithm of FFT is following steps:
1. compute the point value representation of A(x) and B(x) of lengh 20n using the applications of FFT. These representations contain the value of the two polynomial the 20n-th roots of unity,this step takes $O(nlgn)$
2.compute the point value of C(x)=A(x)*B(x) by multiply these values, this step takes $O(n)$
3. Create coefficient representation of C(x) through inverse FFT, this step takes $O(nlgn)$.
So the final result is:

$$T(n) = \theta(nlgn) + \theta(nlgn) = \theta(nlgn)$$

3 QUESTION 3

In this problem, $A(x) = \sum_{j=0}^{n-1} a_j x^j$, whose point value is $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ and y_k is $A(x_k)$ and we can get that:

$$y_0 = a_0 + a_1 x_0 + a_2 (x_0)^2 + ... + a_{n-1} (x_0)^{n-1}$$

$$y_1 = a_0 + a_1 x_1 + a_2 (x_1)^2 + ... + a_{n-1} (x_1)^{n-1}$$

$$...$$

$$y_{n-1} = a_0 + a_1 x_{n-1} + a_2 (x_{n-1})^2 + ... + a_{n-1} (x_{n-1})^{n-1}$$

so for this problem,
for $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$
we consider that $x'_0, x'_1, ...x'_{n-1}$
which $x'_0 = 1/x_0, x'_1 = 1/x_1, ..., x'_{n-1} = 1/x_{n-1}$
use x'_k in $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$
we can get that:

$$\begin{aligned}
y'_0 &= A^{rev}(x') \\
&= a_{n-1} + a_{n-2} x'_0 + a_{n-3} (x'_0)^2 + ... + a_0 (x'_0)^{n-1} \\
&= a_{n-1} + a_{n-2} 1/x_0 + a_{n-3} (1/x_0)^2 + ... + a_0 (1/x_0)^{n-1} \\
&= (1/x_0)^{n-1} (a_0 + a_1 (x_0) + a_2 (x_0)^2) + ...a_{n-1} (x_0)^{n-1}) \\
&= (x'_0)^{n-1} y_0
\end{aligned}$$

so we can get that:

$$y'_k = (x'_k)^{n-1} y_k$$

so the point value representation is

$$(x'_0, y'_0), (x'_1, y'_1), ..., (x'_{n-1}, y'_{n-1})$$

where

$$x'_k = 1/x_k, y_k = (x'_k)^{n-1} y_k$$

4 QUESTION 4

We need to prove that $2T(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) + n = \theta(nlgn)$
Assume that the solution is $T(n) = \theta(nlgn)$
So we need to prove that there is $c_1 > 0, c_2 > 0, c_1 nlgn \leq T(n) \leq c_2 nlgn$
Assume that this bound holds for all $m = (\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)$ so we can get that:

$$c_1(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) \leq T(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) \leq c_2(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)$$

first we consider the right side,

$$T(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) \leq c_2(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)$$

from the equations above , we can get that,

$$\begin{aligned} T(n) &\leq 2 * c_2(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) + n \\ &\leq c_2(n + 2lgn)lgn + n \\ &= O(nlgn) \end{aligned}$$

then we consider the left side:

$$c_1(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) \leq T(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)$$

from the equations above, we can get that,

$$\begin{aligned} T(n) &\geq 2 * c_1(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil)lg(\lfloor \frac{n}{2} \rfloor + \lceil logn \rceil) + n \\ &\geq c_1(n + 2lgn)lg(\frac{n}{2}) + n \\ &\geq c_1(n + 2lgn)lg(n) - c_1(n + 2lgn)lg2 + n \\ &= c_1 * nlgn + 2c_1 lgn * lgn - (c_1 - 1)n - 2c_1 * lg2 * lgn \\ &= \Omega(nlgn) \end{aligned}$$

so we finally get that:

$$T(n) = \theta(nlgn)$$

5 QUESTION 5

Multiply $5 * 5$ matrix using k multiplications, which means we recursively multiply $(n/5) * (n/5)$ matrices. So the time complexity is $T(n) = kT(n/5) + \Theta(n^2)$. And using the master method to solve this recurrence.
so $k(1/5)^x = 1, x = log_5 k, \alpha = 2, \beta = 0$
if $log_5 k < 2$, then $T(n) = \theta(n^2)$, and $K < 25$, $T(n) = O(n^{lg7})$
if $log_5 k = 2$, then $T(n) = \theta(n^2 lgn)$, and $K = 25$ then $T(n) = O(n^{lg7})$
if $log_5 k > 2$, and $K > 25$ then $T(n) = O(n^{lg7})$, when $log_5 k < lg7$
when $k < 5^{lg7} \approx 91.675$
Therefore the largest integer $k = 91$.
Therefore $k = 91$ and the running time is $\theta(n^{log_5 21}) = O(n^{2.80})$

6 QUESTION 6

6.1 PART A

To get $O(nlgn)$ complexity, we need to use divide and conquer method. We every time divide the array into two equal length sub-array $X[1...2/n]$ and $x[(2/n)+1...n]$ and then divide every sub-array into two equal length sub-array until every array's length is 2 or 1. if the length is 1, we just return this value. If the length is 2, their are two conditions, if those two value are equal , we just return this value. If the two value are different, we return nothing. Then every time we return a value, we need to check whether this value is a majority value in above array by traversal the whole array.
For example, we have a array [1,2,2,2], we just divide this array into two sub-array[1,2] and [2,2]. In the first array, the two values are different, so we return nothing. For the second array, the two values are equal, so we return this value. So for the array[1,2,2,2], we just check whether 2 is a majority value by traversal the whole array.So we find that 2 is the majority value.
For another example, we have a array [1,2,2,2,3], we first divide this array into two sub-arrays that [1,2,2] and [2,3]. then we divide the first array into [1,2] and [2]. For the [1,2], there are two different value, so we return nothing. For the [2], we return 2. So we check 2 in [1,2,2], we find that 2 is the majority value in [1,2,2]. Then for the [2,3], there are two different values, so we return nothing. So in the [1,2,2,2,3], we just need to traversal this array to check whether 2 is the majority value and we find that 2 is the majority value.
For the correctness,if there are a majority value in this array, it must be a majority value in either one of the two sub-arrays. Because the majority value must appear at least $(n/2+1)$ times. And the terminate condition is the we do the divide lgn times. For the time complexity, every time we divide the problem into two sub-problems and we need $O(1)$ to divide and $O(n)$ to combine the results. So in a word, $T(n) = 2T(\frac{n}{2}) + n$

so $x = 1, \alpha = 1, \beta = 0$
Finally we get that $T(n) = \theta(nlgn)$

6.2 PART B

To solve this problem ,we need to use a very smart algorithm which is Moore voting algorithm.The central idea is that every time we delete two different value in this array. If there exists a majority value, it must be the rest value in this array. What's more, we need to traversal the whole array with this value to make sure it is the majority value because there is a condition that the left value is not the majority value. For example, [1,2,3], the rest value is 3 , and 3 is not the majority value.
we use c to record a candidate value, which is the majority value currently. And f(c) is the times that c appears. Before the traversal, c is null and $f(c) = 0$. In the process, if $f(c) = 0$, which means now there is no candidate value. So we haven't find the majority value up to now.
At the beginning, we assign that $c = A[0], f(c) = 1$
If the current value $A[i] == c$, then $f(c) += 1$, which means we don't find different value, so we only need to sum the same value.
If the current value $A[i] != c$, then $f(c) -= 1$, which means we delete a different pair value and we don't need to do anything with A[i]
When we finish the traversal, and $f(c) != 0$, so the c is the value we want to find and we need again to traversal the whole array to make sure the value is the value we want to find. If $f(c) = 0$, so there is no majority value in this array.
For the correctness, if x is the majority value, so the x must be the left element. And we need to traversal again to make sure that this left value is majority value. So this algorithm is right.
Below is the Java code of this problem:

```
public class Solution {
    public int majority(int[] num) {

        int major=num[0];
        int count = 1;
        for(int i=1; i<num.length;i++){
            if(count==0){
                count++;
                major=num[i];
            }else if(major==num[i]){
                count++;
            }else count--;

        }
        return major;
    }
}
```

For the time complexity,
because we traversal the array one time , so the time complexity is $O(n)$