Fast Fourier Transform

Table of Contents:

Problem 1

1. $r-1 = n - 1 + n - 1 = 2n - 2$

2. For all j (j = 0 through n-1):

$$c_j = \sum_{i=0}^{j} a_i * b_{j-i}$$

3. $\Theta(n^2)$

This is an algorithm to calculate this:

```
For ( i=0; i<n,i++)
    For ( j=0; j<n;j++ )
        c[j+i]=a[i]*b[j];
```

Notice that j + i goes from 0 to 2n-2.

Problem 2

1. $(x_{c,i}, y_{c,i}) = (x_{a,i}, y_{a,i} * y_{b,i})$ for i = 0 through r-1

   $C = \{(x_{a,0}, y_{a,0} * y_{b,0}), (x_{a,1}, y_{a,1} * y_{b,1}), \dots, (x_{a,r-1}, y_{a,r-1} * y_{b,r-1})\}$

2.

$$C = \sum_{i=0}^{2(n-1)} (x_{a,i}, y_{a,i} * y_{b,i})$$

There is 1 multiplication, so the complexity is:

$$C = \sum_{i=0}^{2(n-1)} 1 = 2n-1 = \Theta(n)$$

Problem 3
Question 1

The complexity of this conversion as a function of n is

$O(n^3)$

because the algorithm breaks down into the summation:

$$\sum_{k=0}^{n-1} \sum_{j=0}^{n-1} \left((1) + \sum_{i=0}^{j} (1) \right) =$$

$$\sum_{k=0}^{n-1} \sum_{j=0}^{n-1} (1) + \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} \sum_{i=0}^{j} (1) =$$

$$n^2 + \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} j =$$

$$n^2 + \sum_{k=0}^{n-1} n*(n+1)/2 =$$

$$n^2 + \sum_{k=0}^{n-1} n^2 =$$

$$n^2 + n^3 = O(n^3)$$

Problem 3
Question 2

```
Convert ( {$a_j$}{$x_k$}  {
     { $y_k$ }= 0;
     for ( k = 0 ; k < n ; k++)
               X = 1 ;
               for (j=0 ; j < n ; j ++)
               {         $y_k$ += $a_j$ * X ;
                         X *= $x_k$ ;
               }
}
```

Question 3:

The complexity of this new algorithm is

$$\Theta(n^2)$$

because the algorithm breaks down into the summation:

$$\sum_{k=0}^{n-1} \sum_{j=0}^{n-1} (2) =$$

$$2n^2 = \Theta(n^2)$$

Problem 4

Case for i = 0 and n = 3:

A(x0) = y0 : n = 0

A(x0) = y0 [((x0-x1)/(x0-x1)) *((x0-x2)/(x0-x2)) *((x0-x3)/(x0-x3))] +
Y1 [((x0-x0)/(x1-x1)) *((x0-x2)/(x1-x2)) *((x0-x3)/(x1-x3))] +
y2 [((x0-x0)/(x2-x0)) *((x0-x1)/(x2-x1)) *((x0-x3)/(x2-x3))]

A(x0) = y0 [(1) *(1) *(1)] +
Y1 [(0) *(0) *(0)] +
Y2 [(0) *(0) *(0)]

A(x0) = y0

$$A(x_i) = \sum_{k=0}^{n-1} y_k \prod_{\substack{j=0 \\ k \,!= j}}^{n-1} ((x_i-x_j)/(x_k-x_j))$$

Split out case i.

$$A(x_i) = y_i \prod_{\substack{j = 0 \\ j \,!= i}}^{n-1} ((x_i-x_j)/(x_i-x_j)) + \sum_{\substack{k=0 \\ k \,!= i}}^{n-1} y_k \prod_{\substack{j=0 \\ j \,!= k}}^{n-1} ((x_i-x_j)/(x_k-x_j))$$

In the second part, one piece of each summation will = 0.

Because at some point j=i (x_i-x_j) will = 0 and so the entire second part =0 and will drop out.

$$y_k \prod_{\substack{j=0 \\ j \,!= k}}^{n-1} ((x_i-x_j)/(x_k-x_j)) = 0$$

The first part will be simply y_i because the mult-ation = 1.

$$y_i \prod_{\substack{j=0 \\ j \,!= i}}^{n-1} ((x_i-x_j)/(x_i-x_j)) = y_i \prod_{\substack{j=0 \\ j \,!= i}}^{n-1} (1) = y_i$$

Conditions:

x_i != x_j for all i != j

No 2 x's in the vector are the same, that way you are never dividing by zero.
This is naturally true for point value representation because you need n-1 distinct points, therefore xk are all distinct.

Problem 5:

The complexity of steps

$a + b + c = \Theta(n^2) + \Theta(n) + \Theta(n^2) = \Theta(n^2)$

This growth rate is the same as the growth rate of the algorithm in problem 1.

note: The complexity of Lagrange's formula is $\Theta(n^2)$.

Problem 6

Proof 1

Prove that : $(q_{n,k}+n/2)^2 = (q_{n,k})^2$

by definition:

$$(q_{n,k})^2 = (e^{2\pi ik/n})^2$$

and:

$$(q_{n,k}+n/2)^2 = ((e^{2\pi i*(k+n/2))(1/n)})^2$$

$$= e^{2\pi i ((2k+n)/n)}$$

$$= e^{2\pi i (2k/n+1)}$$

$$= e^{2\pi i (2k/n)} * e^{2\pi i}$$

$$= e^{2 \pi i (2k/n)}$$

$$= (e^{2 \pi i (k/n)})^2$$

$$= (q_{n,k})^2$$

Problem 6

Proof 2

Prove that : $(q_{n,k})^2 = q_{n/2,k}$

by definition:

$q_{n/2,k} \quad = e^{(2\pi\, ik)/(n/2)}$

$\qquad = e^{(4\pi\, i\, k)/n}$

$\qquad = (e^{2\,\pi\, i\, k/n})^2$

$\qquad = (q_{n,k})^2$

Problem 6 Question 3:

There are **n/2** distinct numbers

This is because $(q_{n,k+n/2})^2 = (q_{n,k})^2$

While there are n distinct $q_{n,k}$,
When they are squared, they form n/2 pairs of equal nth roots of unity,
or just n/2 distinct numbers.

Problem 6 Question 4: For this question, I created three algorithms. Version 1 is the most straight forward Version. Version 3 is the most optimized version.

Problem 6 Question 4:Version 1
In this version, each x ($q_{n,0}$, $q_{n,1}$, ... , $q_{n,n-1}$) is evaluated at each level, and an array of values are passed up (returned recursively), until an array of all the final values is developed. Note that this version creates new arrays for the returned values, and the newly generated values to be returned to the previous recursion.

Also, note that the

```
Poly2Point( A[],n ) {
        if (n=1) { return A[]; }

        for(i=0 ; i<n/2 ; i++) {  // split into A0 and A1
        A0[i]=A[2i];
        A1[i]=A[2i+1];
        }
        Yodd[]=Poly2Point(A0,n/2); // Yodd = A0($x^2$)
        Yeven[]=Poly2Point(A1,n/2);  // Yeven = A1($x^2$)
        q=1;
        $q_{n,1} = e^{(2\pi i/n)}$;
        for(k=0 ; k<n ; k++) {   // $q_{2,1}=q_{4,1}^2 = q_{4,3}^2 = q_{8,1}^4 = q_{8,5}^4 ...$
                Y[k]=Yeven[k%(n/2)] + q * Yodd[k%(n/2)];   //  q = $q_{n,k}$
                q = q * $q_{n,1}$;
        }
        return Y[];
}
```

Problem 6 Question 4: Version 2:

In this second version, I use the property

$q_{n,k+(n/2)} = -q_{n,k}$

{ which can be derived from the fact that :

$$(q_{n,k+n/2})^2 = (q_{n,k})^2$$

but $q_{n,k+n/2} \;!=\; q_{n,k}$ }

to reduce the number of iterations in the for loop from n to n/2.

```
Poly2Point( A[],n ) {
        if (n=1) { return A[]; }

        g = n/2;
        for(i=0 ; i<g ; i++) {  // split into A0 and A1
        A0[i]=A[2i];
        A1[i]=A[2i+1];
        }
        Yodd[]=Poly2Point(A0,g); // Yodd = A0(x^2)
        Yeven[]=Poly2Point(A1,g);  // Yeven = A1(x^2)
        q=1;
        $q_{n,1} = e^{(2\pi i/n)}$;
        for(k=0 ; k<g ; k++) {   // $q_{2,1}=q_{4,1}^2 = q_{4,3}^2 = q_{8,1}^4 = q_{8,5}^4 ...$
            Y[k]=Yeven[k] + q * Yodd[k];
            Y[k+g] = Yeven[k] - q * Yodd[k];
            q = q * $q_{n,1}$;
        }
        return Y[];
}
```

Problem 6 Question 4: Version 3:

In this third version, I speed up the algorithm by avoiding the unnecessary creation of arrays. To do this, I exploit the fact that A0 and Yodd are the same size, and A[] and Y[] are the same size. Plus, once A[] is split into A0[] and A1[], A[] is no longer needed. Therefore, it can be used so store Y and passed back to the previous recursion by reference. Note that once the recursion is don, A0 and A1 go out of scope and the memory reclaimed. Therefore, no extra arrays need to be established "newed up" or deleted.

All we have to do is pass A[] by reference and the next recursion can use it.
Then nothing has to be returned.

```
Poly2Point( & A[],n ) {  // A[] is called by reference
        if (n=1) { return ; }

        g = n/2;
        for(i=0 ; i<g ; i++) {  // split into A0 and A1
        A0[i]=A[2i];
        A1[i]=A[2i+1];
        }
        Poly2Point(A0,g); // A0 = A0($x^2$) after call.
        Poly2Point(A1,g);  // A1 = A1($x^2$) after call.
        q=1;
        $q_{n,1} = e^{(2\pi i/n)}$;

        for(k=0 ; k<g ; k++) {   // $q_{2,1}=q_{4,1}^2 = q_{4,3}^2 = q_{8,1}^4 = q_{8,5}^4 \ldots$
            A[k]=A0[k] + q * A1[k];
            A[k+g] = A0[k] - q * A1[k];
            q = q * $q_{n,1}$;
        }
        return ;  // A[] was passed by reference from a previous recursion.
                  // It may have been A0[] or A1[] in the previous recursion.
                  // It is implicitly passed back.
}
```

Problem 6 Question 5:

Algorithm analysis:Version 3

Since N is halved each iteration,

The number of recursions is $\log_2 N$.

Plus, 3N/2 multiplications are performed at each level because there are 3 multiplications in the for loop, and it loops N/2 times.

$T(N) = O(N \log_2 N)$ (Growth Rate)

Because

$F(n) = O(C) + 2F(n/2)$,

Let $O(C) = 3N/2$, because there are 3 multiplications in the for loop, and it loops N/2 times.

$F(0) = 0$
$F(1) = 0 + 2F(0) = 0$ // base case
$F(2) = 3N/2 + 2F(1) = 3 + 0 = 3$
$F(4) = 3N/2 + 2F(2) = 6 + 6 = 12$
$F(8) = 3N/2 + 2F(4) = 12 + 24 = 36$
$F(16) = 3N/2 + 2F(8) = 24 + 72 = 96$
$F(32) = 3N/2 + 2F(16) = 48 + 192 = 240$
$F(N) = 3N/2 + 2F(N/2) = 3N/2 * (\log_2 N) = O(N\log_2 N)$

i.e. $F(32) = 3(32)/2*(\log_2 32) = 48*(5) = 240$

This is also a lower bound, because even if extra orders are filled in with zero's to make it a 2^k order, the multiplications still take place.

Therefore, $F(N) = \Theta(N\log_2 N)$

Problem 7:

The complexity of steps

a + b + c using the problem 6 algorithm =

$\Theta\ (n(\log_2 n)) + \Theta\ (n) + \Theta\ (n(\log_2 n)) = \Theta\ (n(\log_2 n))$