Fast Fourier Transform

Different steps of FFT are:

- Convert coefficient value form to point value form
  $Y_k = A(x_k)$
  Where $A(x) = \Sigma a_i x^i$
  Now to find the point value-form of $A(x)$ of degree n-1, we need n distinct points.
  Now, in fft, these n distinct points will be w_n^k where k=0 ..., n-1
  Therefore
  $$A(x_k) = A(w_n^k) = \Sigma a_i w_n^{ki} \text{ where } i=0...n-1$$

- So let there be a polynomial A(x) with degree n−1, where n is a power of 2, and n>1:
  $$A(x)=a_0x^0+a_1x^1+\dots+a_{n-1}x^{n-1}$$
  We divide it into two smaller polynomials, the one containing only the coefficients of the even positions, and the one containing the coefficients of the odd positions:
  $$A_0(x)=a_0x^0+a_2x^1+\dots+a_{n-2}x^{n/2-1}$$
  $$A_1(x)=a_1x^0+a_3x^1+\dots+a_{n-1}x^{n/2-1}$$
  It is easy to see that
  $$A(x)=A_0(x^2)+xA_1(x^2).$$
  The basic idea of the FFT is to apply **divide and conquer**. We divide the coefficient vector of the polynomial into two vectors, recursively compute the even and odd polynomial for each of them, and combine the results.

- Since we are evaluating the A(x) at n distinct roots of unity, we must have evaluated $A(x^2)$ at $(w_n^0)^2$, $(w_n^1)^2$, $(w_n^2)^2$.... $(w_n^{n-1})^2$

  Then $A_0(x^2)$ and $A_1(x^2)$ must have been evaluated on n/2 distinct points of unity where n/2 distinct points must have been $(w_n^0)^2$, $(w_n^1)^2$, $(w_n^2)^2$.... $(w_n^{n/2-1})^2$ by using lemma 3
  I.e,
  Now we also know that
  $$w_n^{2k} = w_{n/2}^k$$
  We can say that $(w_n^0)^2$, $(w_n^1)^2$, $(w_n^2)^2$.... $(w_n^{n/2-1})^2 = w_{n/2}^0,\ w_{n/2}^1, \dots\ w_{n/2}^{n/2-1}$
  Hence $A_0(x^2)$ and $A_1(x^2)$ are evaluated on $w_{n/2}^0,\ w_{n/2}^1, \dots\ w_{n/2}^{n/2-1}$

- $A(x)=A_0(x^2)+xA_1(x^2)$ then
  $$A(w_n^k) = A_0(w_n^k)^2 + w_n^k A_1(w_n^k)^2 \text{ for } k = 0 ,\dots n\text{-}1$$
  $$A(w_n^k) = A_0(w_{n/2}^k) + w_n^k A_1(w_{n/2}^k) \quad \text{for } k=0 \dots n/2\text{-}1 \text{ as } w_n^{2k} = w_{n/2}^k$$

  Also,

$$A(w_n^{(k+n/2)}) = A_0(w_n^{(k+n/2)})^2 + w_n^{(k+n/2)} A_1 (w_n^{(k+n/2)})^2$$

From lemma 4,

$$A(w_n^{(k+n/2)}) = A_0(w_{n/2}^{k}) + w_{n/2}^{k} A_1 (w_{n/2}^{k})$$

From lemma 5

$$A(w_n^{(k+n/2)}) = A_0(w_{n/2}^{k}) - w_{n/2}^{k} A_1 (w_{n/2}^{k})$$

Hence for k = 0 …. n/2-1

$$A(w_n^{k}) = A_0(w_{n/2}^{k}) + w_n^{k} A_1(w_{n/2}^{k})$$

$$A(w_n^{(k+n/2)}) = A_0(w_{n/2}^{k}) - w_{n/2}^{k} A_1 (w_{n/2}^{k})$$

Pseudocode

```
//   create a complex class as follows
class complex {

        double realPart;
        double complexPart;

        //   parameterized constructor
        complex(double realPart , double complexPart){
                this.realPart = realPart ;
                this.complexPart = complexPart;
        }

        //   default constructor
        complex(){
                realPart = 0 ;
                complexPart = 0;
        }
}
```

```
 double PI = Math.acos(-1);        //   assign a global variable PI as cos-1 -1

function FFT() {
        /*
```

```
                    create a complex type array A of the polynomial with real part as the
                              coefficient of each term and imaginary part as 0 and add 4
complex values to it.
                    Example: array below represents the polynomial
                    A(x) = 1 + 2x + 3x^2 + 4x^3

                    complex a[] = {new complex(1, 0), new complex(2, 0), new complex(3, 0),
new                           complex(4, 0)};
          */
          complex[] omega = init_omega(A.length);

          complex[] y = fft(A, omega);

}

function static complex[] fft(complex[] A, complex[] omega) {

          n = A.length ;
          if(n == 1)
                    return A ;

          half = n >> 1 ;
          complex[] Aeven = new complex[half];
          complex[] Aodd = new complex[half];
          i=0
          j=0

          while( i < n ) {
                    Aeven[j] = A[i];
                    Aodd[j] = A[i+1];
                    i = i+2
                    j=j+1
          }

          //   recursive calls
          complex[] yeven = fft(Aeven, omega);
          complex[] yodd = fft(Aodd, omega);

          complex[] yn = new complex[n] ;

          /*
                    calculating A(x) by
                    A(w_n^k) = A_0(w_{n/2}^k) +w_n^k A_1(w_{n/2}^k)
```

```
                    A(w_n^(k + n/2)) = A_0 (w_{n/2}^k )  -  w_{n/2}^k A_1 (w_{n/2}^k)

          */
          for(int k = 0 ; k < half ; k++) {
                    complex multiply = multiplycomplex(omega[k], yodd[k]);
                    complex add = addcomplex(yeven[k], multiply);
                    complex subtract = subtractcomplex(yeven[k], multiply);
                    yn[k] = add ;
                    yn[k+half] = subtract;
          }


          return yn;


}

//   function to multiply complex numbers with return type as complex
function complex multiplycomplex(complex a, complex b) {

          complex nc = new complex();

          nc.realPart += (a.realPart*b.realPart) ;
          nc.realPart -= (a.complexPart*b.complexPart) ;

          nc.complexPart += (a.realPart*b.complexPart);
          nc.complexPart += (a.complexPart*b.realPart);

          return nc ;


}

//   function to subtract complex numbers with return type as complex
function complex subtractcomplex(complex a, complex b) {

          return new complex(a.realPart-b.realPart, a.complexPart-b.complexPart);
}

//   function to add complex numbers with return type as complex
function complex addcomplex(complex a, complex b) {

          return new complex(a.realPart+b.realPart, a.complexPart + b.complexPart);
}
```

```
/*
	function to calculate n roots of unity i,e: Wn^0, Wn^1, Wn^2, Wn^3, .... Wn^(n-1) given an n
	where each root is a complex number, so the return type is complex
*/
public static complex[] init_omega(int n){

	complex[] omega = new complex[n];
	double angle = 2*(PI/n);


	for(int k = 0 ; k < n ; k++) {

		omega[k] = new complex(Math.cos(angle*k), Math.sin(angle*k));
	}


	return omega;

}
```