CSC 2400, Fall 2013
Programming Assignment #2
Assigned: November 15, 2013
Due: December 4, 2013
Weight: 12%

The goal of this assignment is to implement and compare various algorithms for evaluating polynomials with complex coefficients at the n^{th} roots of unity.

The n^{th} roots of unity are $e^{2\pi ik/n}$, where i is the complex number representing the square root of negative 1, for k = 0, …, n - 1. If you raise each of the n^{th} roots of unity to the n^{th} power, you will get 1. A complex number can be represented as a + b i, where a and b are real numbers and can be drawn on the 2D plane at the coordinate (a,b). Euler's Theorem showed that $e^{i\theta} = \cos\theta + i \sin\theta$. θ is an angle expressed in radians (remember that 360 degrees equals 2π radians), and $e^{i\theta}$ represents a point on the unit circle (radius 1) centered at (0,0).

Your program will have the capability of asking the user for choices until he/she wishes to exit. A GUI for your program is permitted, but not required. The choices should allow the following:

1. Generate a random polynomial of degree n-1, where all coefficients are of the form (a + 0 i) and a is a whole number with absolute value at most 10. The value of n will be determined by user input. Error checking should be performed on n to make sure it is positive.
2. Read the coefficients of a polynomial of degree n-1 from an input file. The first line will have the number n, and the next n lines will have the coefficients for the terms of degrees 0 through n-1, respectively. These coefficients will be complex coefficients, with the real part first and separated from the complex part by a comma. For example, 5 + 3 i would be represented as a line with 5,3 on it. Error checking should be performed on n to make sure it is positive.
3. Write the coefficients of a polynomial of degree n-1 to an output file. The format will be the same as the format described in the previous option.
4. Run the naïve algorithm for polynomial evaluation on the polynomial that has most recently been generated, either randomly or read from a file. If a polynomial has not been generated yet, inform the user. We covered the naïve algorithm

back when we covered Chapter 3 on brute force and exhaustive search. The algorithm will be run for each of the n^{th} roots of unity.

5. Run Horner's algorithm for polynomial evaluation on the polynomial that has most recently been generated. If a polynomial has not been generated yet, inform the user. Horner's algorithm was also covered in Chapter 3. The algorithm will also be run for each of the n^{th} roots of unity.
6. Run a variation of the naïve algorithm, where the exponentiation is performed by repeated squaring, a decrease-by-half algorithm covered in class in Chapter 4, on the polynomial that has most recently been generated. If a polynomial has not been generated yet, inform the user. The algorithm will also be run for each of the n^{th} roots of unity.
7. Run the Fast Fourier Transform algorithm on the polynomial that has most recently been generated. If a polynomial has not been generated yet, inform the user. The algorithm runs once and evaluates the polynomial at all n^{th} roots of unity. Pseudocode is included at the end of this document. The Fast Fourier Transform is a famous algorithm with applications in digital music and lots of other areas.
8. Display the **running times** for the four algorithms that you have implemented. As usual, error checking should be performed. For Java programmers, you need to consult the System class API to find a suitable method. For programmers in other languages, you need to consult a library with timing functions/methods.
9. Display the counts of the numbers of **complex number multiplications** for the four algorithms that you have implemented.

In implementing the algorithms, it may be helpful to define a Complex struct/class and implement addition, subtraction, and multiplication operations/methods. The fields of the Complex struct/class should be doubles because of the calculations with the n^{th} roots of unity.

It may also be helpful to write two versions of each of your algorithms: one to do the actual work, and one to do the work augmented with the counting operations.

You should follow standard conventions for programming and documentation style. You will submit a zip file of your source code (with compilation instructions). The zip file must include graphs displaying your results for #8 and #9 above when run for n = 4, 8, 16,

32, 64, 128, 256, 512, 1024, and 2048. These graphs must be neatly drawn with suitable software.

Pseudocode for Fast Fourier Transform

```
fft(n, a_0,a_1,…,a_n-1) {
    if (n == 1) return a_0

    (e_0,e_1,…,e_n/2-1) ← FFT(n/2, a_0,a_2,a_4,…,a_n-2)
    (d_0,d_1,…,d_n/2-1) ← FFT(n/2, a_1,a_3,a_5,…,a_n-1)

    for k = 0 to n/2 - 1 {
        ω^k ← e^2πik/n
        y_k     ← e_k + ω^k d_k
        y_k+n/2 ← e_k - ω^k d_k
    }

    return (y_0,y_1,…,y_n-1)
}
```