CSE 417: Algorithms and Computational Complexity
Assignment #3
January 23, 2004
due: Wednesday, February 4

In this assignment you will implement two different solutions to the polynomial multipoint evaluation problem (based on Horner's rule and the FFT) and compare their CPU times.

1. Implement the abstract data type (class) `Complex`. This should contain the following methods:

   (a) `Complex plus(Complex a, Complex b); /* returns a+b */`

   (b) `Complex minus(Complex a, Complex b); /* returns a-b */`

   (c) `Complex times(Complex a, Complex b); /* returns a*b */`

   (d) `Complex root(int n); /* returns a primitive n-th root of unity */`

   Read Section 12.4.3 for a review of the definitions of these complex operations. In particular, that section gives the primitive n-th root of unity in the form $\cos(2\pi/n) + i\sin(2\pi/n)$, which is a much easier way for you to compute it than the equivalent formula $e^{2\pi i/n}$.

2. Implement a procedure

   ```
   void powers(int n, Complex[] omega);
   ```

   that initializes the array `omega` described in the "Remark" in Algorithm 12.4.

3. Implement the recursive procedure `recursiveFFT` given in Algorithm 12.4.

4. Implement a procedure

   ```
   Complex horner(Complex[] P, int n, Complex x);
   ```

   that takes the polynomial `P` of degree `n-1` with Complex coefficients, evaluates it at the value `x`, and returns the result. Your procedure should use Horner's rule to do this evaluation.

5. Now generate a single random polynomial of each of the following degrees: 1, 3, 7, 15, 31, 63, ... Each coefficient should be either 0 or 1, each with probability 1/2. For each of these polynomials P of degree $n-1$, evaluate P at the first n powers of a primitive n-th root of unity, once using `recursiveFFT` and once using n invocations of `horner`. Keep doubling n until you run out of memory or you don't have the patience to wait for the Horner version to finish, but at a minimum you must continue until the Horner version takes more time than the FFT version.

   (a) Use your programming language's processor time facility (in C it is the `clock` function) to measure the elapsed processor time used by each of these two methods.[1] Plot these elapsed times together on a single graph, where the x axis is $\log_2 n$ (where $n-1$ is the polynomial degree) and the y axis is $\log_2 t$ (where t is the elapsed time). Do your graphs make sense to you, given what we showed about these algorithms' running times? Notice the value of n where FFT becomes faster than Horner's rule.

   (b) It would be nice if you could compare the outputs of the FFT and Horner method to be sure they produce the same answers; this would be a good check that your polynomial evaluation procedures don't have bugs. But because of roundoff errors, these numbers will not be exactly identical. If the FFT produces the complex numbers $F_0, F_1, \ldots, F_{n-1}$ and Horner's rule produces the complex numbers $H_0, H_1, \ldots, H_{n-1}$, compute the value

$$\text{diff}(n) = \max(|F_0 - H_0|, |F_1 - H_1|, \ldots, |F_{n-1} - H_{n-1}|),$$

   where $|a + bi| = \max(|a|, |b|)$. If these values diff($n$) are not very small, then you have a bug in one of your procedures.

Turn in your source code, your graphs from part 5(a), and your list of diff(n) values from part 5(b).

[1] Suppose that you are trying to do a particular function invocation F whose elapsed processor time is too small to measure accurately. (To be on the safe side, let's say that any elapsed time less than 500 milliseconds is too small to be measured accurately, since the clock ticks about every 10 milliseconds.) F could be either a call to "horner" or to "recursiveFFT" with a smallish value of n. To get a timing of F, use a simple loop that repeats R times the exact same computation F, where R is chosen large enough so that the elapsed time of these R identical computations is at least 500 milliseconds. Measure the total elapsed time of the R invocations, and divide by R. This gives you the time for a single invocation of F. Of course, you will have to make R bigger the smaller n is.