CSE 521: The Design and Analysis of Algorithms
Assignment #2 (FFT miniproject)
April 16, 1997
due: Wednesday, April 30

Instructions: For this miniproject on the fast Fourier transform, I would like you to work in teams of two. If that is impractical for some reason, please see me.

Overview: In implementing an algorithm such as the fast Fourier transform that works for many different fields (or other algebras), it makes sense to encapsulate the field elements together with the appropriate field operators as an abstract data type or object, and apply the algorithm to this encapsulated type. In this miniproject, you will select a programming language that is good for supporting such abstraction, and implement a small package for polynomial manipulation. The motivation for this is that it is the way that polynomials might be handled in a symbolic manipulation package such as Mathematica or Maple.

(Anecdotal warning: the first time I assigned this miniproject, nearly every team that chose `C++` as their programming language eventually decided that there was no clean way to do the abstraction in that language. The teams that chose other appropriate languages voiced no such regrets.)

Abstract data types for polynomials and fields: Let F be an arbitrary field that supports the fast Fourier transform, i.e., F has the appropriate primitive roots of unity. Your goal is to supply a package that supports the ring $F[x]$, i.e., it provides the five ring operators $+$, $-$, 0, $*$, and 1 for polynomials in $F[x]$. These operators should be encapsulated in a "polynomial" data type.

In order to do this, you will need an underlying abstract data type for the field F, which itself provides the field operators $+$, $-$, 0, $*$, $^{-1}$, and 1 for elements of F. The body of each operator for $F[x]$ will invoke some of the operators for the data type for F. For instance, the $+$ operator of $F[x]$ simply adds the polynomials term by term, using the $+$ operator of F to add coefficients. The bodies of the operators for F will only be supplied when this data type is instantiated to a particular field (described below).

Implement the $*$ operator of $F[x]$ using the evaluation-interpolation scheme discussed in lecture. This involves implementing the FFT and FFI algorithms from Lipson, which should also be written in terms of the abstract field F and your abstract data types above.

For the $F[x]$ abstract data type, you should include procedures that input and output polynomials from and to the user. Do this in any way that you find convenient, provided the output is readable.

For the F abstract data type, you should include procedures that input and output elements of F (for use by the analogous procedures of $F[x]$), and a function that takes a positive integer argument N and returns a primitive Nth root of unity. Again, the bodies of these procedures will be supplied at the time that F is instantiated.

Instantiation: Provide two instantiations of the abstract data type for F, one for $\mathcal{Z}_p$, the integers modulo an arbitrary prime p, and one for $\mathcal{C}$, the complex numbers.

For $\mathcal{Z}_p$, the field operators are all straightforward except for $^{-1}$. To compute a^{-1}, read Section 33.2 in Cormen, Leiserson, and Rivest and use EXTENDED-EUCLID(a, p). (See the proof of Theorem 33.13.) The procedure to compute a primitive Nth root of unity ω in $\mathcal{Z}_p$ is beyond the scope of this assignment, and you should feel free to "cheat" and simply have it ask the user to supply ω or look it up in a very limited table that you build in. If you want roots of unity for relatively large values of p, you can use Figure 2 on page 306 of Lipson as follows: for any line of this table and

any $m \leq e$, $\alpha^{(p-1)/(2^m)} \bmod p$ is a primitive 2^mth root of unity in $\mathcal{Z}_p$. If you need more help with primitive roots, ask me. (If you are ambitious or simply interested in how to compute primitive roots of unity without "cheating", read Section IX.1.3 of Lipson.)

Notice that, to get a concrete prime size field, you must instantiate the abstract data type for $\mathcal{Z}_p$ once more by providing a value of p.

For $\mathcal{C}$, $(a+bi)^{-1} = \frac{a}{a^2+b^2} - \frac{b}{a^2+b^2}i$. A primitive Nth root of unity is $\cos\frac{2\pi}{N} + i\sin\frac{2\pi}{N}$.

Experimentation with roundoff error: Test your package by manipulating polynomials with integer coefficients. (Include in your tests some polynomial multiplications that are small enough that you can check your output by hand.) Perform each polynomial computation once by embedding it in $\mathcal{C}$ and once by embedding it in $\mathcal{Z}_p$ for a sufficiently large prime p. (You need only ensure that p is large enough that no coefficient in the answer exceeds $(p-1)/2$ in absolute value. This ensures that there is no "wraparound" in the output coefficients. For instance, if you multiply two polynomials each of degree at most n, you are certainly safe if $p > 2nB^2$, where B is the maximum absolute value of any input coefficient.)

Now for the grand challenge: Experiment a little to see how bad you can make the roundoff error when you embed the computation in $\mathcal{C}$. Use double precision. Can you find a computation in which some output coefficient is off by at least 1? You can use your answers from the $\mathcal{Z}_p$ embedding as a check, since they will be perfectly accurate, provided you choose p as described in the previous paragraph. Is there a way to choose the coefficients of two input polynomials so that their product has some bad coefficient? Is there some more complicated computation involving other polynomial operations that causes bad roundoff? Keep in mind that one way to produce roundoff error is to compute the difference of two extremely close numbers, or add a large number to one very close to zero, and then use the result as a factor in some product.

Limitations: Don't worry too much about aspects of your implementation that I haven't discussed, unless you are enjoying it. As an example, I'm not concerned about whether your implementation permits arbitrarily large degree polynomials, or arbitrarily large coefficients, unless this somehow interferes with your ability to experiment with roundoff error. If in doubt about whether some aspect is important, please ask.

What to turn in: I would like you to turn in the following items:

1. a listing of your program and testing,
2. a brief description of any neat encapsulation features of your program or your chosen programming language that make your implementation particularly interesting, or reasons why the programming language turned out to be unsatisfactory, and
3. a brief description of what you discovered about roundoff error when the fast Fourier transform is computed over $\mathcal{C}$.