At this stage each of the Fourier transforms in (9.82) uses only one data point. We see from (9.78) with $N = 1$ that the value of each of these Fourier transforms, $g_k^{eee}, g_k^{eeo}, \ldots$, is equal to the value of f at the corresponding data point. Note that for $N = 8$, we have performed $3 = \log_2 N$ decompositions. In general, we would perform $\log_2 N$ decompositions.

There are two steps to the FFT. First, we reorder the components so that they appear in the order given in (9.82). This step makes the subsequent calculations easier to organize. To see how to do the reordering, we rewrite (9.82) using the values of f:

$$\begin{aligned} g_k = f(0) &+ W^{4k} f(4\Delta) + W^{2k} f(2\Delta) + W^{6k} f(6\Delta) \\ &+ W^{k} f(\Delta) + W^{5k} f(5\Delta) + W^{3k} f(3\Delta) + W^{7k} f(7\Delta). \end{aligned} \tag{9.83}$$

We use a trick to obtain the ordering in (9.83) from the original order $f(0\Delta), f(1\Delta), \ldots, f(7\Delta)$. Part of the trick is to refer to each g in (9.82) by a string of "e" and "o" characters. We assign 0 to "e" and 1 to "o" so that each string represents the binary representation of a number. If we reverse the order of the representation, that is, set 110 to 011, we obtain the value of f we want. For example, the fifth term in (9.82) contains g^{oee}, corresponding to the binary number 100. The reverse of this number is 001, which equals 1 in decimal notation, and hence the fifth term in (9.83) contains the function $f(1\Delta)$. Convince yourself that this bit reversal procedure works for the other seven terms.

The first step in the FFT algorithm is to use this bit reversal procedure on the original array representing the data. In the next step this array is replaced by its Fourier transform. If you want to save your original data, it is necessary to first copy the data to another array before passing the array to a FFT implementation. The `SimpleFFT` class implements the Danielson–Lanczos algorithm using three loops. The outer loop runs over $\log_2 N$ steps. For each of these steps, N calculations are performed in the two inner loops. As can be seen in Listing 9.13, in each pass through the innermost loop, each element of the array `g` is updated once by the quantity `temp` formed from a power of W multiplied by the current value of an appropriate element of `g`. The power of W used in `temp` is changed after each pass through the innermost loop. The power of the FFT algorithm is that we do not separately multiply each $f(j\Delta)$ by the appropriate power of W. Instead, we first take pairs of $f(j\Delta)$ and multiply them by an appropriate power of W to create new values for the array `g`. Then we repeat this process for pairs of the new array elements (each array element now contains four of the $f(j\Delta)$). We repeat this process until each array element contains a sum of all N values of $f(j\Delta)$ with the correct powers of W multiplying each term to form the Fourier transform.

Listing 9.13 A simple implementation of the FFT algorithm.

```
package org.opensourcephysics.sip.ch09;
public class SimpleFFT {
   public static void transform(double[] real, double[] imag) {
      int N = real.length;
      int pow = 0;
      while(N/2>0) {
         if(N%2==0) { // N should be even
            pow++;
            N /= 2;
         } else {
            throw new IllegalArgumentException("Number of points in
                 this FFT implementation must be even.");
         }
      }
```

```
      int N2 = N/2;
      int jj = N2;
      // rearrange input according to bit reversal
      for(int i = 1;i<N-1;i++) {
         if(i<jj) {
            double tempRe = real[jj];
            double tempIm = imag[jj];
            real[jj] = real[i];
            imag[jj] = imag[i];
            real[i] = tempRe;
            imag[i] = tempIm;
         }
         int k = N2;
         while(k<=jj) {
            jj = jj-k;
            k = k/2;
         }
         jj = jj+k;
      }
      jj = 1;
      for(int p = 1;p<=pow;p++) {
         int inc = 2*jj;
         double wp1 = 1, wp2 = 0;
         double theta = Math.PI/jj;
         double cos = Math.cos(theta), sin = -Math.sin(theta);
         for(int j = 0;j<jj;j++) {
            for(int i = j;i<N;i += inc) {
               // calculate the transform of 2^p
               int ip = i+jj;
               double tempRe = wp1*real[ip]-wp2*imag[ip];
               double tempIm = wp2*real[ip]+wp1*imag[ip];
               real[ip] = real[i]-tempRe;
               imag[ip] = imag[i]-tempIm;
               real[i] = real[i]+tempRe;
               imag[i] = imag[i]+tempIm;
            }
            double temp = wp1;
            wp1 = wp1*cos-wp2*sin;
            wp2 = temp*sin+wp2*cos;
         }
         jj = inc;
      }
   }
}
```

Exercise 9.41 Testing the FFT algorithm

1. Test the `SimpleFFT` class for $N = 8$ by going through the code by hand and showing that the class reproduces (9.83).
2. Display the Fourier coefficients of random real values of $f(j\Delta)$ for $N = 8$ using both `SimpleFFT` and the direct computation of the Fourier coefficients based on (9.34). Compare the two sets of data to insure that there are no errors in `SimpleFFT`. Repeat for a random collection of complex data points.