Computing the vector $(Y_0, Y_1, \ldots, Y_{N-1})$ using formula (8.5) requires

$$(N-1)^2 \text{ complex multiplications,}$$
$$N(N-1) \text{ complex additions,}$$

assuming that the values of ω_N^j, the sines and cosines of the given angles, have already been computed and stored.

A typical value of N is of the order of 1000, which implies about a million operations of each kind. Considering the frequency of this computation, it was natural to seek to lower the cost. In 1965, two American scientists, J. W. Cooley and J. W. Tukey, developed a much more efficient algorithm that has since been known as the *fast Fourier transform* (FFT). This algorithm takes into consideration the special form of the transformation matrix, which is constructed from the roots of unity. From the beginning, the FFT, including its many extensions, has enjoyed enormous success. In fact, it is safe to say that it has been the backbone of signal and image processing in the last half of the twentieth century. Furthermore, it has been the inspiration for numerous investigations in algebra independently of its intensive use in signal processing. It was indeed a marvelous discovery. The fast Fourier transform marked an important step in the theory of the *complexity of algorithm.* This field of research is concerned with determining and minimizing the cost of a given computation or class of computations, where the cost is measured by the number of numerical operations. For example, we will see that the cost of the FFT is of the order $N \log N$.

9.1 The Cooley–Tukey algorithm

Assume that N is even, $N = 2m$, and rearrange the terms of (8.5) into two groups—those with even indices and those with odd indices. Then

$$Y_k = \frac{1}{2}\,(P_k + \omega_N^{-k} I_k),$$

where The P_k and I_k are given by the formulas

$$P_k = \frac{1}{m}\left(y_0 + y_2\omega_N^{-2k} + \cdots + y_{N-2}\omega_N^{-(N-2)k}\right),$$
$$I_k = \frac{1}{m}\left(y_1 + y_3\omega_N^{-2k} + \cdots + y_{N-1}\omega_N^{-(N-2)k}\right).$$

Note that we have the relations

$$P_{k+m} = P_k, \quad I_{k+m} = I_k, \quad \omega_N^{-(k+m)} = -\omega_N^{-k}$$

for $k = 0, 1, \ldots, m-1$. These identities provide the key to the algorithm, whose essential idea is this:

Step 1: Compute P_k and $\omega_N^{-k} I_k$;

Step 2: Form $Y_k = \frac{1}{2}(P_k + \omega_N^{-k} I_k)$;

Step 3: Deduce $Y_{k+m} = \frac{1}{2}(P_k - \omega_N^{-k} I_k)$.

These computations are done successively only for $k = 0, 1, \ldots, m-1$. This scheme is illustrated schematically in Figure 9.1, where the arrows indicate dependent relations in the calculations.

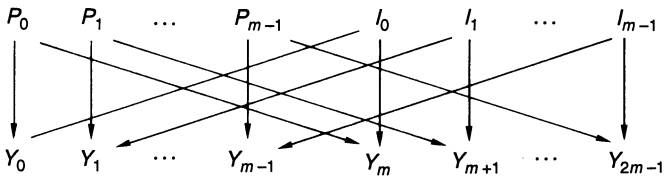

FIGURE 9.1.

The cost of Step 1 is $2(m-1)^2+m-1$ (or roughly $N^2/2$) multiplications. Steps 2 and 3 cost nothing in complex multiplications. Thus one obtains the same result for about half the work. (Note we have neglected the divisions by 2 and m. In practice, m is a power of 2, and these are binomial shifts.)

One could consider that this saving is sufficient and stop here. But most readers probably have noticed that P_k and I_k are themselves two independent discrete Fourier transforms of order $m = N/2$. In any case, it takes only a moment to be convinced that

$$(y_0, y_2, \ldots, y_{2m-2}) \overset{\mathcal{F}_{N/2}}{\longmapsto} (P_0, P_1, \ldots, P_{m-1}),$$
$$(y_1, y_3, \ldots, y_{2m-1}) \overset{\mathcal{F}_{N/2}}{\longmapsto} (I_0, I_1, \ldots, I_{m-1}).$$

An obvious strategy is to repeat this clever decomposition, provided that m is even. The best case is where N is a power of 2, $N = 2^p$. We can then iterate the process until we arrive at discrete Fourier transforms of order 2. These are particularly simple computations, since they are of the form

$$Y = (y + z)/2,$$
$$Z = (y - z)/2.$$

We illustrate the algorithm for $N = 8$. The first step is to rearrange the sequence $(y_1, y_2, \ldots, y_8)$ into two sequences of length 4, the first having the odd indices and the second the even indices. The process is repeated, and we obtain the four vectors of length 2 shown in Figure 9.2. The computation begins with the vectors of length 2. As in Figure 9.1, the arrows indicate

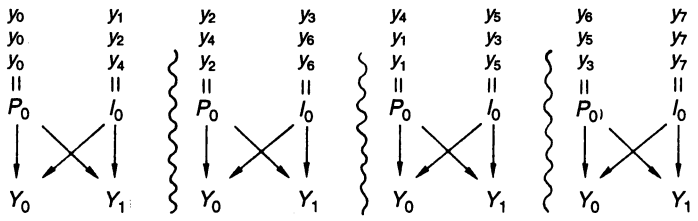

FIGURE 9.2. Rearrangement of the data.

the dependencies of the Y-vectors on the data. To simplify the notation, we have written P_0, I_0, Y_0, Y_1 four times, but they are clearly not the same values. The wiggly lines separate independent computations. Going from one level (vectors of length m) to the next (vectors of length $2m$) is done using the formulas

$$Y_k = \frac{1}{2}(P_k + \omega_N^{-k} I_k),$$
$$Y_{k+m} = \frac{1}{2}(P_k - \omega_N^{-k} I_k),$$

for $k = 0, 1, \ldots, m-1$. Figure 9.3 illustrates the complete algorithm for $N = 2^3$. The wiggly lines separate the independent computations. In an actual program, a single vector is used. This is ultimately the output vector $(Y_0, Y_1, \ldots, Y_{N-1})$; it is the result of successively transforming the vector obtained by appropriately rearranging the original data.

9.2 Evaluating the cost of the algorithm

The only arithmetic operations that appear in the FFT are multiplications and additions of complex numbers. (We neglect the successive divisions by 2; these reduce to a single division by $N = 2^p$, at the outset, for example.) We denote the cost of r complex multiplications and of s complex additions by $[r; s]$.

For $N = 2^p$, let M_p be the number of multiplications used in the algorithm and let A_p be the number of additions. Formulas (9.1) are used to evaluate the cost for $N = 2^p$ in terms of the cost for $N = 2^{p-1}$:

Cost of computing the P_k: $[M_{p-1}; A_{p-1}]$;
Cost of computing the I_k: $[M_{p-1}; A_{p-1}]$;

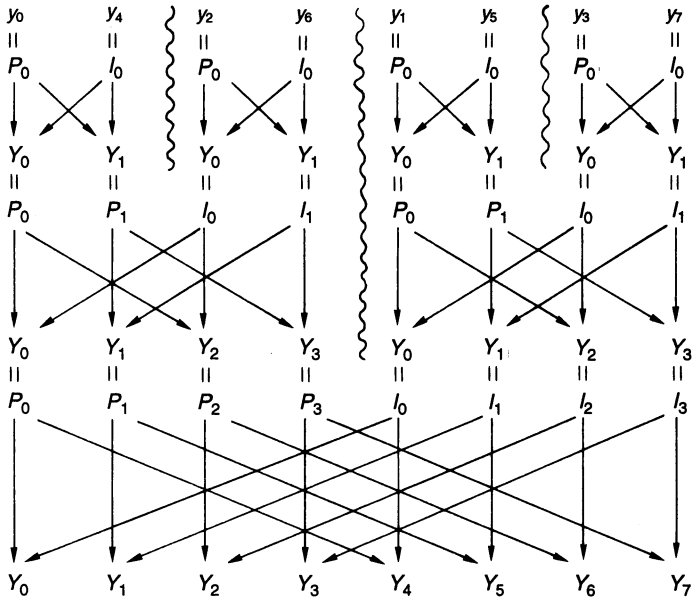

FIGURE 9.3. The FFT algorithm of order 8.

Multiplications by ω_N^{-k} $(k \geq 1)$: $[2^{p-1} - 1;\ 0]$;
Additions: $[0;\ 2^p]$.

From these relations we have

$$M_1 = 0, \qquad A_1 = 2,$$
$$M_p = 2M_{p-1} + 2^{p-1} - 1, \qquad A_p = 2A_{p-1} + 2^p.$$

A computation, which is left as an exercise, provides an explicit expressions for M_p and A_p, namely,

$$M_p = (p-2)2^{p-1} + 1,$$
$$A_p = p2^p.$$

We see from this that the global cost, as a function of N, is

$$\left[\frac{1}{2}N(\log_2 N - 2) + 1;\ N\log_2 N\right]. \tag{9.1}$$

Table 9.1 compares the FFT with the "old" method. It shows the savings for the two operations as a function of N. For $N = 1024$, we see that the FFT divides the cost by 250: a fantastic gain.

9.3 The mirror permutation

If we wish to obtain the values $Y_0, Y_1, \ldots, Y_{N-1}$ in this order, it is clear from Figure 9.3 that we must begin with a vector (y_n), $n = p(k)$, where

	Multiplications			Additions		
N	Old Method	FFT	Ratio	Old Method	FFT	Ratio
2	0	0		2	2	1
4	0	0		12	8	1.5
8	49	5	10	56	24	2.3
16	225	17	13	240	64	3.8
32	961	49	20	992	160	6.2
64	3,969	129	31	4,032	384	10
128	16,129	321	50	16,256	896	18
256	65,025	769	85	65,280	2,048	32
512	261,121	1,793	145	261,632	4,608	57
1,024	1,046,529	4,097	255	1,047,552	10,240	102

TABLE 9.1.

p is a permutation of the indices $k = 0, 1, \ldots, N-1$. This permutation of the data at the outset is an important issue, particularly for programming the algorithm. There are a number of ways to do this, and it is a problem that generally stimulates much imagination from students. The only restriction is not to introduce so many operations that the gain realized by the algorithm is compromised.

For these consecutive even–odd permutations, one feels that the representation of the indices in base 2 must come into play. Take the case $N = 8$ and notice what happens:

$$\begin{array}{l||l} 0 = 000 \cdot\cdot & \cdot\cdot\ 0 = 000 \\ 1 = 001 \cdot\cdot & \cdot\cdot\ 4 = 100 \\ 2 = 010 \cdot\cdot & \cdot\cdot\ 2 = 010 \\ 3 = 011 \cdot\cdot & \cdot\cdot\ 6 = 110 \\ 4 = 100 \cdot\cdot & \cdot\cdot\ 1 = 001 \\ 5 = 101 \cdot\cdot & \cdot\cdot\ 5 = 101 \\ 6 = 110 \cdot\cdot & \cdot\cdot\ 3 = 011 \\ 7 = 111 \cdot\cdot & \cdot\cdot\ 7 = 111 \end{array}$$

Each number has been written in binary form using three places, which is possible, since we stop at 7 ($N = 2^3$). We notice a surprising property: The required permutation of an index is given by reversing the order of its binary representation. It is as if they were reflected in a mirror. One can verify that this holds for $N = 16$, and it is an excellent exercise to show that it is true in general.

This "mirror" permutation leads to a method for programming the initial permutation. For this, it is necessary to work with the binary representations of the indices. These, however, are not directly accessible in high-level languages like PASCAL; consequently, this is not the best method.

9.4 A recursive program

Here, to finish the chapter, is a program (written in a simplified pseudo-language) for computing the FFT of a vector y. It is taken from [Lip81]. We include it because it is astonishingly simple to program and because it follows step by step the approach we have taken. The particularity of this procedure is that it is *recursive*, which means that calls are made within the program to the program itself.

```
Procedure FFT(n,w,y,Y);
    begin
        if n=1 then Y[0]:=y[0] else
        begin
            m:=n div 2;
            for k:=0 to m−1 do
            begin
                b[k]:=y[2*k];
                c[k]:=y[2*k+1]
            end; w2=w*w;
            TFR(m,w2,b,B);
            TFR(m,w2,c,C);
            wk:=1;
            for k:=0 to m−1 do
            begin
                X:=B[k]; T:=wk*C[k];
                Y[k]:=(X+T)/2;
                Y[k+m]:=(X−T)/2;
                wk=wk*w
            end
        end
    end.
```

We note that compilers deal with these recursions more or less well, particularly on microcomputers. While the program itself is concisely written, which is very attractive to the programmer, its execution, by contrast, requires a great deal of processing and a large amount of memory: At each call to FFT, the procedure is completely recopied with new parameters.

Finally, it is not obvious that this procedure does indeed compute the desired FFT. For example, the second call to FFT is executed only after many other such calls.