

transform using the Goertzel algorithm, the number of real multiplications required is approximately N^2 and the number of real additions is approximately $2N^2$. While this is more efficient than the direct computation of the discrete Fourier transform, the amount of computation is still proportional to N^2 .

In either the direct method or the Goertzel method we do not need to evaluate $X[k]$ at all N values of k . Indeed, we can evaluate $X[k]$ for any M values of k , with each DFT value being computed by a recursive system of the form of Figure 9.2 with appropriate coefficients. In this case, the total computation is proportional to NM . The Goertzel method and the direct method are attractive when M is small; however, as indicated previously, algorithms are available for which the computation is proportional to $N \log_2 N$ when N is a power of 2. Therefore, when M is less than $\log_2 N$, either the Goertzel algorithm or the direct method may in fact be the most efficient method, but when all N values of $X[k]$ are required, the decimation-in-time algorithms, to be considered next, are roughly $(N/\log_2 N)$ times more efficient than either the direct method or the Goertzel method.

9.3 DECIMATION-IN-TIME FFT ALGORITHMS

In computing the DFT, dramatic efficiency results from decomposing the computation into successively smaller DFT computations. In this process, we exploit both the symmetry and the periodicity of the complex exponential $W_N^{kn} = e^{-j(2\pi/N)kn}$. Algorithms in which the decomposition is based on decomposing the sequence $x[n]$ into successively smaller subsequences are called *decimation-in-time algorithms*.

The principle of the decimation-in-time algorithm is most conveniently illustrated by considering the special case of N an integer power of 2, i.e., $N = 2^\nu$. Since N is an even integer, we can consider computing $X[k]$ by separating $x[n]$ into two $(N/2)$ -point³ sequences consisting of the even-numbered points in $x[n]$ and the odd-numbered points in $x[n]$. With $X[k]$ given by

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, \dots, N-1, \quad (9.10)$$

and separating $x[n]$ into its even- and odd-numbered points, we obtain

$$X[k] = \sum_{n \text{ even}} x[n] W_N^{nk} + \sum_{n \text{ odd}} x[n] W_N^{nk}, \quad (9.11)$$

or, with the substitution of variables $n = 2r$ for n even and $n = 2r + 1$ for n odd,

$$\begin{aligned} X[k] &= \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k} \\ &= \sum_{r=0}^{(N/2)-1} x[2r] (W_N^2)^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1] (W_N^2)^{rk}. \end{aligned} \quad (9.12)$$

³When discussing FFT algorithms, we use the words *sample* and *point* interchangeably to mean *sequence value*. Also, we refer to a sequence of length N as an N -point sequence, and the DFT of a sequence of length N will be called an N -point DFT.

But $W_N^2 = W_{N/2}$, since

$$W_N^2 = e^{-2j(2\pi/N)} = e^{-j2\pi/(N/2)} = W_{N/2}. \tag{9.13}$$

Consequently, Eq. (9.12) can be rewritten as

$$\begin{aligned} X[k] &= \sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk} \\ &= G[k] + W_N^k H[k], \quad k = 0, 1, \dots, N-1. \end{aligned} \tag{9.14}$$

Each of the sums in Eq. (9.14) is recognized as an $(N/2)$ -point DFT, the first sum being the $(N/2)$ -point DFT of the even-numbered points of the original sequence and the second being the $(N/2)$ -point DFT of the odd-numbered points of the original sequence. Although the index k ranges over N values, $k = 0, 1, \dots, N-1$, each of the sums must be computed only for k between 0 and $(N/2) - 1$, since $G[k]$ and $H[k]$ are each periodic in k with period $N/2$. After the two DFTs are computed, they are combined according to Eq. (9.14) to yield the N -point DFT $X[k]$. Figure 9.3 depicts this computation for $N = 8$. In this figure, we have used the signal flow graph conventions that were introduced in Chapter 6 for representing difference equations. That is, branches entering a node are summed to produce the node variable. When no coefficient is indicated, the branch transmittance is assumed to be unity. For other branches, the transmittance of a branch is an integer power of W_N .

In Figure 9.3, two 4-point DFTs are computed, with $G[k]$ designating the 4-point DFT of the even-numbered points and $H[k]$ designating the 4-point DFT of the odd-numbered points. $X[0]$ is then obtained by multiplying $H[0]$ by W_N^0 and adding the product to $G[0]$. $X[1]$ is obtained by multiplying $H[1]$ by W_N^1 and adding that result to $G[1]$. Equation (9.14) states that, to compute $X[4]$, we should multiply $H[4]$ by W_N^4 and add the result of $G[4]$. However, since $G[k]$ and $H[k]$ are both periodic in k with

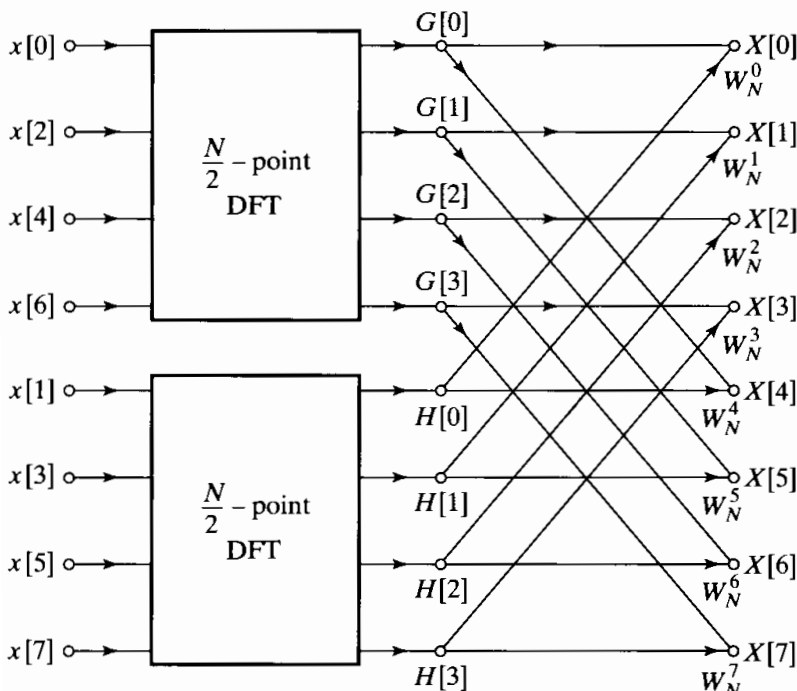


Figure 9.3 Flow graph of the decimation-in-time decomposition of an N -point DFT computation into two $(N/2)$ -point DFT computations ($N = 8$).

period 4, $H[4] = H[0]$ and $G[4] = G[0]$. Thus, $X[4]$ is obtained by multiplying $H[0]$ by W_N^4 and adding the result to $G[0]$. As shown in Figure 9.3, the values $X[5]$, $X[6]$, and $X[7]$ are obtained similarly.

With the computation restructured according to Eq. (9.14), we can compare the number of multiplications and additions required with those required for a direct computation of the DFT. Previously we saw that, for direct computation without exploiting symmetry, N^2 complex multiplications and additions were required.⁴ By comparison, Eq. (9.14) requires the computation of two $(N/2)$ -point DFTs, which in turn requires $2(N/2)^2$ complex multiplications and approximately $2(N/2)^2$ complex additions if we do the $(N/2)$ -point DFTs by the direct method. Then the two $(N/2)$ -point DFTs must be combined, requiring N complex multiplications, corresponding to multiplying the second sum by W_N^k , and N complex additions, corresponding to adding the product obtained to the first sum. Consequently, the computation of Eq. (9.14) for all values of k requires at most $N + 2(N/2)^2$ or $N + N^2/2$ complex multiplications and complex additions. It is easy to verify that for $N > 2$, the total $N + N^2/2$ will be less than N^2 .

Equation (9.14) corresponds to breaking the original N -point computation into two $(N/2)$ -point DFT computations. If $N/2$ is even, as it is when N is equal to a power of 2, then we can consider computing each of the $(N/2)$ -point DFTs in Eq. (9.14) by breaking each of the sums in that equation into two $(N/4)$ -point DFTs, which would then be combined to yield the $(N/2)$ -point DFTs. Thus, $G[k]$ in Eq. (9.14) would be represented as

$$G[k] = \sum_{r=0}^{(N/2)-1} g[r]W_{N/2}^{rk} = \sum_{\ell=0}^{(N/4)-1} g[2\ell]W_{N/2}^{2\ell k} + \sum_{\ell=0}^{(N/4)-1} g[2\ell+1]W_{N/2}^{(2\ell+1)k}, \quad (9.15)$$

or

$$G[k] = \sum_{\ell=0}^{(N/4)-1} g[2\ell]W_{N/4}^{\ell k} + W_{N/2}^k \sum_{\ell=0}^{(N/4)-1} g[2\ell+1]W_{N/4}^{\ell k}. \quad (9.16)$$

Similarly, $H[k]$ would be represented as

$$H[k] = \sum_{\ell=0}^{(N/4)-1} h[2\ell]W_{N/4}^{\ell k} + W_{N/2}^k \sum_{\ell=0}^{(N/4)-1} h[2\ell+1]W_{N/4}^{\ell k}. \quad (9.17)$$

Consequently, the $(N/2)$ -point DFT $G[k]$ can be obtained by combining the $(N/4)$ -point DFTs of the sequences $g[2\ell]$ and $g[2\ell+1]$. Similarly, the $(N/2)$ -point DFT $H[k]$ can be obtained by combining the $(N/4)$ -point DFTs of the sequences $h[2\ell]$ and $h[2\ell+1]$. Thus, if the 4-point DFTs in Figure 9.3 are computed according to Eqs. (9.16) and (9.17), then that computation would be carried out as indicated in Figure 9.4. Inserting the computation of Figure 9.4 into the flow graph of Figure 9.3, we obtain the complete flow graph of Figure 9.5, where we have expressed the coefficients in terms of powers of W_N rather than powers of $W_{N/2}$, using the fact that $W_{N/2} = W_N^2$.

For the 8-point DFT that we have been using as an illustration, the computation has been reduced to a computation of 2-point DFTs. For example, the 2-point DFT of the sequence consisting of $x[0]$ and $x[4]$ is depicted in Figure 9.6. With the computation

⁴For simplicity, we shall assume that N is large, so that $(N-1)$ can be approximated by N .

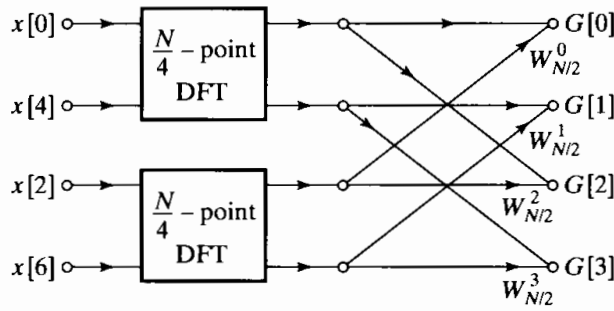


Figure 9.4 Flow graph of the decimation-in-time decomposition of an $(N/2)$ -point DFT computation into two $(N/4)$ -point DFT computations ($N = 8$).

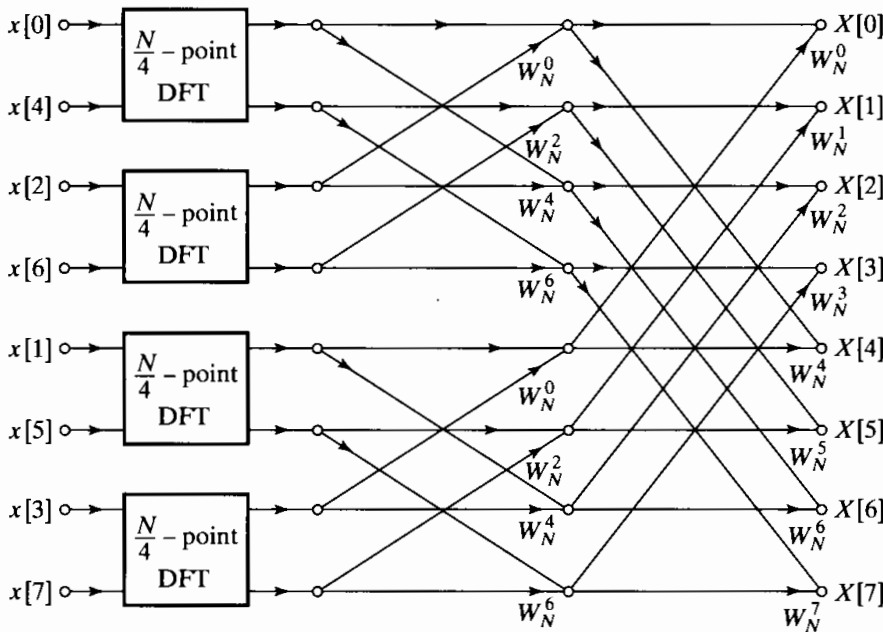


Figure 9.5 Result of substituting the structure of Figure 9.4 into Figure 9.3.

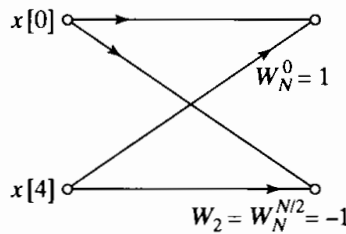


Figure 9.6 Flow graph of a 2-point DFT.

of Figure 9.6 inserted in the flow graph of Figure 9.5, we obtain the complete flow graph for computation of the 8-point DFT, as shown in Figure 9.7.

For the more general case, but with N still a power of 2, we would proceed by decomposing the $(N/4)$ -point transforms in Eqs. (9.16) and (9.17) into $(N/8)$ -point transforms and continue until we were left with only 2-point transforms. This requires $\nu = \log_2 N$ stages of computation. Previously, we found that in the original decomposition of an N -point transform into two $(N/2)$ -point transforms, the number of complex multiplications and additions required was $N + 2(N/2)^2$. When the $(N/2)$ -point transforms are decomposed into $(N/4)$ -point transforms, the factor of $(N/2)^2$ is replaced by $N/2 + 2(N/4)^2$, so the overall computation then requires $N + N + 4(N/4)^2$ complex multiplications and additions. If $N = 2^\nu$, this can be done at most $\nu = \log_2 N$ times, so that after carrying out this decomposition as many times as possible, the number of complex multiplications and additions is equal to $N\nu = N\log_2 N$.

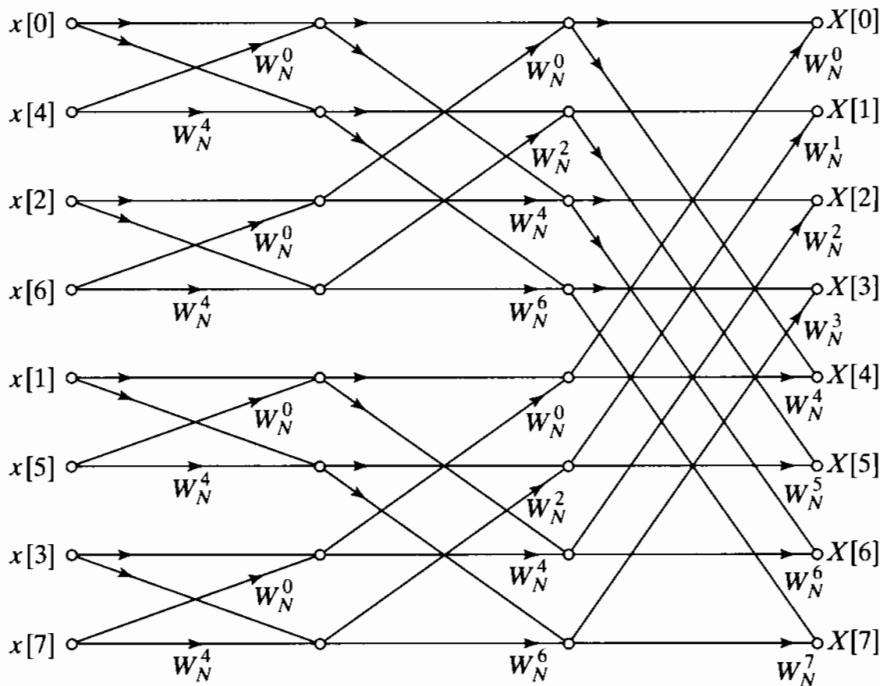


Figure 9.7 Flow graph of complete decimation-in-time decomposition of an 8-point DFT computation.

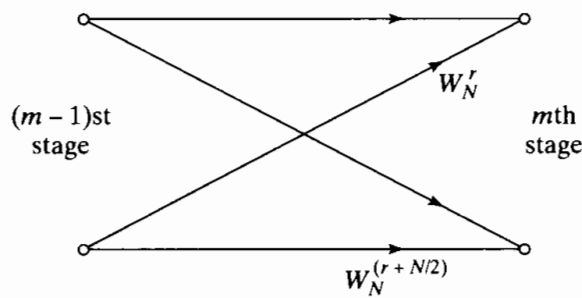


Figure 9.8 Flow graph of basic butterfly computation in Figure 9.7.

The flow graph of Figure 9.7 displays the operations explicitly. By counting branches with transmittances of the form W_N^r , we note that each stage has N complex multiplications and N complex additions. Since there are $\log_2 N$ stages, we have a total of $N \log_2 N$ complex multiplications and additions. This is the substantial computational savings that we have previously indicated was possible. For example, if $N = 2^{10} = 1024$, then $N^2 = 2^{20} = 1,048,576$, and $N \log_2 N = 10,240$, a reduction of more than two orders of magnitude!

The computation in the flow graph of Figure 9.7 can be reduced further by exploiting the symmetry and periodicity of the coefficients W_N^r . We first note that, in proceeding from one stage to the next in Figure 9.7, the basic computation is in the form of Figure 9.8, i.e., it involves obtaining a pair of values in one stage from a pair of values in the preceding stage, where the coefficients are always powers of W_N and the exponents are separated by $N/2$. Because of the shape of the flow graph, this elementary computation is called a *butterfly*. Since

$$W_N^{N/2} = e^{-j(2\pi/N)N/2} = e^{-j\pi} = -1, \tag{9.18}$$

the factor $W_N^{r+N/2}$ can be written as

$$W_N^{r+N/2} = W_N^{N/2} W_N^r = -W_N^r. \tag{9.19}$$

With this observation, the butterfly computation of Figure 9.8 can be simplified to the form shown in Figure 9.9, which requires only one complex multiplication instead of two. Using the basic flow graph of Figure 9.9 as a replacement for butterflies of the form of Figure 9.8, we obtain from Figure 9.7 the flow graph of Figure 9.10. In particular, the number of complex multiplications has been reduced by a factor of 2 over the number in Figure 9.7.

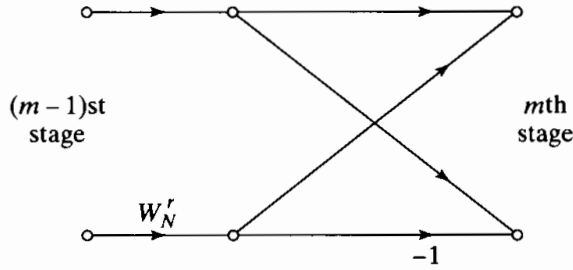


Figure 9.9 Flow graph of simplified butterfly computation requiring only one complex multiplication.

9.3.1 In-Place Computations

The flow graph of Figure 9.10 describes an algorithm for the computation of the discrete Fourier transform. The essential features of the flow graph are the branches connecting the nodes and the transmittance of each of these branches. No matter how the nodes in the flow graph are rearranged, it will always represent the same computation, provided that the connections between the nodes and the transmittances of the connections are maintained. The particular form for the flow graph in Figure 9.10 arose out of deriving the algorithm by separating the original sequence into the even-numbered and odd-numbered points and then continuing to create smaller and smaller subsequences in the same way. An interesting by-product of this derivation is that this flow graph, in addition to describing an efficient procedure for computing the discrete Fourier transform, also suggests a useful way of storing the original data and storing the results of the computation in intermediate arrays.

To see this, it is useful to note that according to Figure 9.10, each stage of the computation takes a set of N complex numbers and transforms them into another set of

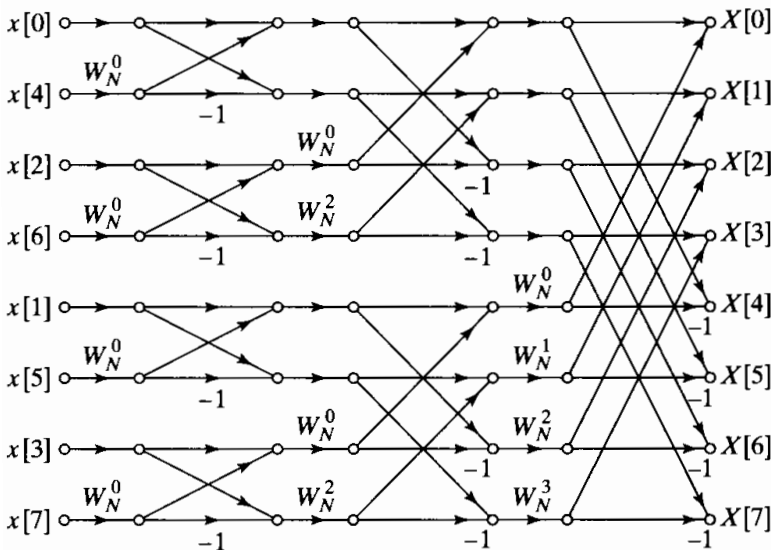


Figure 9.10 Flow graph of 8-point DFT using the butterfly computation of Figure 9.9.

N complex numbers through basic butterfly computations of the form of Figure 9.9. This process is repeated $\nu = \log_2 N$ times, resulting in the computation of the desired discrete Fourier transform. When implementing the computations depicted in Figure 9.10, we can imagine the use of two arrays of (complex) storage registers, one for the array being computed and one for the data being used in the computation. For example, in computing the first array in Figure 9.10, one set of storage registers would contain the input data and the second set would contain the computed results for the first stage. While the validity of Figure 9.10 is not tied to the order in which the input data are stored, we can order the set of complex numbers in the same order that they appear in the figure (from top to bottom). We denote the sequence of complex numbers resulting from the m th stage of computation as $X_m[\ell]$, where $\ell = 0, 1, \dots, N - 1$, and $m = 1, 2, \dots, \nu$. Furthermore, for convenience, we define the set of input samples as $X_0[\ell]$. We can think of $X_{m-1}[\ell]$ as the input array and $X_m[\ell]$ as the output array for the m th stage of the computations. Thus, for the case of $N = 8$, as in Figure 9.10,

$$\begin{aligned}
 X_0[0] &= x[0], \\
 X_0[1] &= x[4], \\
 X_0[2] &= x[2], \\
 X_0[3] &= x[6], \\
 X_0[4] &= x[1], \\
 X_0[5] &= x[5], \\
 X_0[6] &= x[3], \\
 X_0[7] &= x[7].
 \end{aligned}
 \tag{9.20}$$

Using this notation, we can label the input and output of the butterfly computation in Figure 9.9 as indicated in Figure 9.11, with the associated equations

$$X_m[p] = X_{m-1}[p] + W_N^r X_{m-1}[q], \tag{9.21a}$$

$$X_m[q] = X_{m-1}[p] - W_N^r X_{m-1}[q]. \tag{9.21b}$$

In Eqs. (9.21), p, q , and r vary from stage to stage in a manner that is readily inferred from Figure 9.10 and from Eqs. (9.11), (9.14), (9.16), etc. It is clear from Figures 9.10 and 9.11 that only the complex numbers in locations p and q of the $(m - 1)$ st array are required to compute the elements p and q of the m th array. Thus, only one complex array of N storage registers is physically necessary to implement the complete computation if $X_m[p]$ and $X_m[q]$ are stored in the same storage registers as $X_{m-1}[p]$ and $X_{m-1}[q]$, respectively. This kind of computation is commonly referred to as an *in-place*

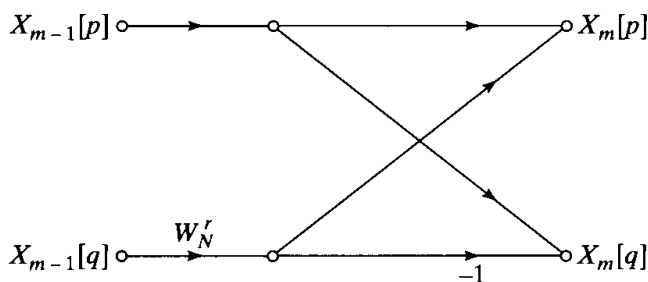


Figure 9.11 Flow graph of Eqs. (9.21).

computation. The fact that the flow graph of Figure 9.10 (or Figure 9.7) represents an in-place computation is tied to the fact that we have associated nodes in the flow graph that are on the same horizontal line with the same storage location and the fact that the computation between two arrays consists of a butterfly computation in which the input nodes and the output nodes are horizontally adjacent.

In order that the computation may be done in place as just discussed, the input sequence must be stored (or at least accessed) in a nonsequential order, as shown in the flow graph of Figure 9.10. In fact, the order in which the input data are stored and accessed is referred to as *bit-reversed* order. To see what is meant by this terminology, we note that for the 8-point flow graph that we have been discussing, three binary digits are required to index through the data. Writing the indices in Eqs. (9.20) in binary form, we obtain the following set of equations:

$$\begin{aligned}
 X_0[000] &= x[000], \\
 X_0[001] &= x[100], \\
 X_0[010] &= x[010], \\
 X_0[011] &= x[110], \\
 X_0[100] &= x[001], \\
 X_0[101] &= x[101], \\
 X_0[110] &= x[011], \\
 X_0[111] &= x[111].
 \end{aligned}
 \tag{9.22}$$

If (n_2, n_1, n_0) is the binary representation of the index of the sequence $x[n]$, then the sequence value $x[n_2, n_1, n_0]$ is stored in the array position $X_0[n_0, n_1, n_2]$. That is, in determining the position of $x[n_2, n_1, n_0]$ in the input array, we must reverse the order of the bits of the index n .

Let us first consider the process depicted in Figure 9.12 for sorting a data sequence in normal order by successive examination of the bits representing the data index. If the most significant bit of the data index is zero, $x[n]$ belongs in the top half of the

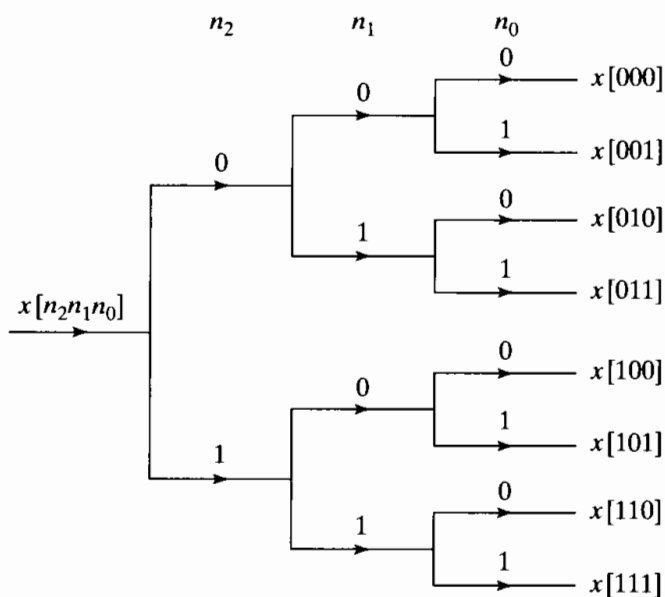


Figure 9.12 Tree diagram depicting normal-order sorting.

sorted array; otherwise it belongs in the bottom half. Next, the top half and bottom half subsequences can be sorted by examining the second most significant bit, and so on.

To see why bit-reversed order is necessary for in-place computation, recall the process that resulted in Figure 9.7 and Figure 9.10. The sequence $x[n]$ was first divided into the even-numbered samples, with the even-numbered samples occurring in the top half of Figure 9.3 and the odd-numbered samples occurring in the bottom half. Such a separation of the data can be carried out by examining the least significant bit $[n_0]$ in the index n . If the least significant bit is 0, the sequence value corresponds to an even-numbered sample and therefore will appear in the top half of the array $X_0[\ell]$, and if the least significant bit is 1, the sequence value corresponds to an odd-numbered sample and consequently will appear in the bottom half of the array. Next, the even- and odd-indexed subsequences are sorted into their even- and odd-indexed parts, and this can be done by examining the second least significant bit in the index. Considering first the even-indexed subsequence, if the second least significant bit is 0, the sequence value is an even-numbered term in the subsequence, and if the second least significant bit is 1, then the sequence value has an odd-numbered index in this subsequence. The same process is carried out for the subsequence formed from the original odd-indexed sequence values. This process is repeated until N subsequences of length 1 are obtained. This sorting into even- and odd-indexed subsequences is depicted by the tree diagram of Figure 9.13.

The tree diagrams of Figures 9.12 and 9.13 are identical, except that for normal sorting, we examine the bits representing the index from left to right, whereas for the sorting leading naturally to Figure 9.7 or 9.10, we examine the bits in reverse order, right to left, resulting in bit-reversed sorting. Thus, the necessity for bit-reversed ordering of the sequence $x[n]$ results from the manner in which the DFT computation is decomposed into successively smaller DFT computations in arriving at Figures 9.7 and 9.10.

9.3.2 Alternative Forms

Although it is reasonable to store the results of each stage of the computation in the order in which the nodes appear in Figure 9.10, it is certainly not necessary to do so. No matter how the nodes of Figure 9.10 are rearranged, the result will always be

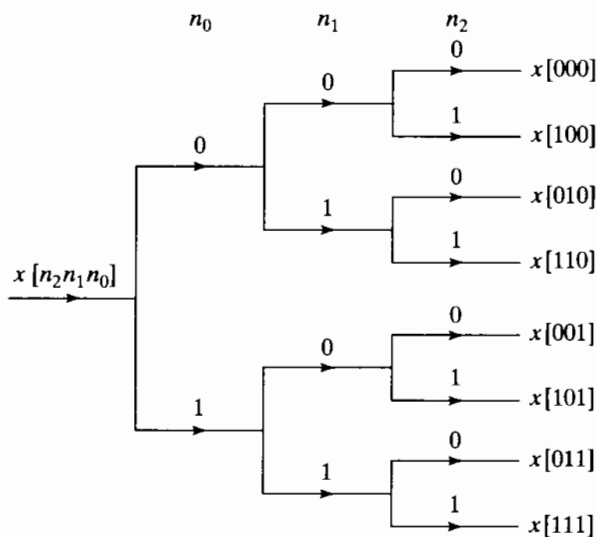


Figure 9.13 Tree diagram depicting bit-reversed sorting.