Introduction to

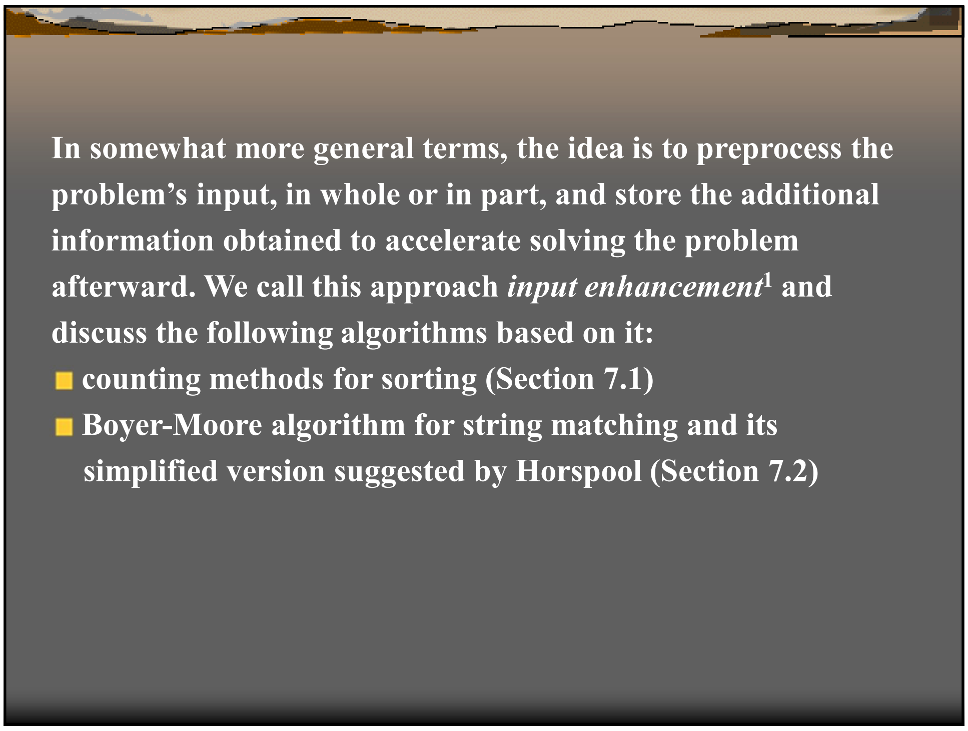
The Design and Analysis of Algorithms

3rd Edition

7

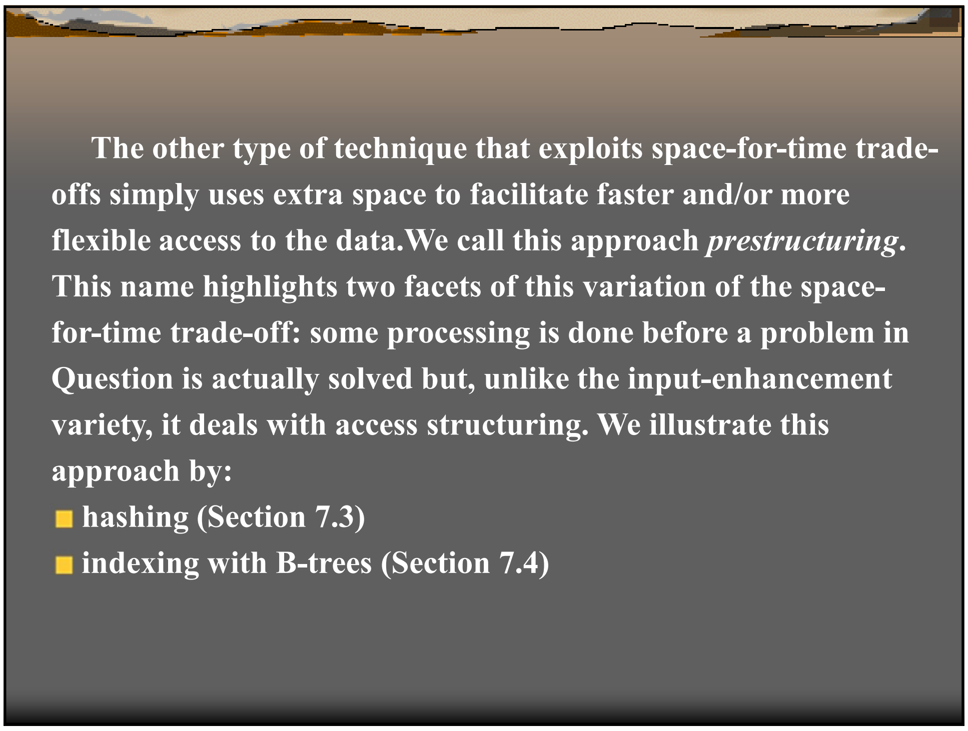
Space and Time Trade-Offs

Space and time trade-offs in algorithm design are a well-known issue for both theoreticians and practitioners of computing. Consider, as an example, the problem of computing values of a function at many points in its domain. If it is time that is at a premium, we can precompute the function's values and store them in a table. This is exactly what human computers had to do before the advent of electronic computers, in the process burdening libraries with thick volumes of mathematical tables. Though such tables have lost much of their appeal with the widespread use of electronic computers, the underlying idea has proven to be quite useful in the development of several important algorithms for other problems.



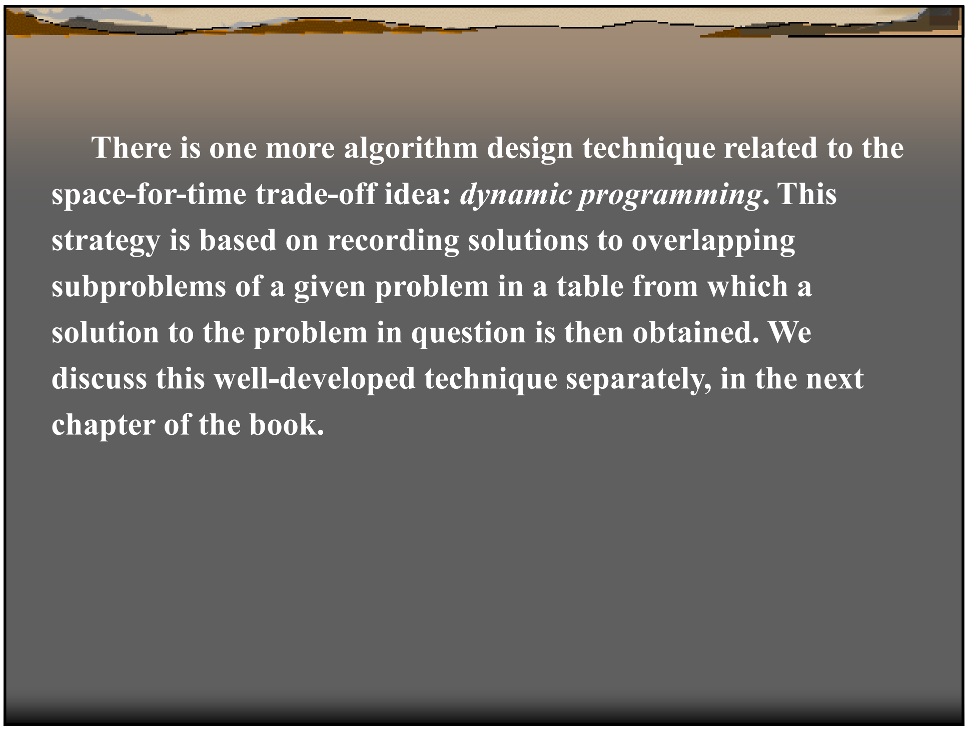
In somewhat more general terms, the idea is to preprocess the problem's input, in whole or in part, and store the additional information obtained to accelerate solving the problem afterward. We call this approach *input enhancement*[1] and discuss the following algorithms based on it:

- counting methods for sorting (Section 7.1)
- Boyer-Moore algorithm for string matching and its simplified version suggested by Horspool (Section 7.2)

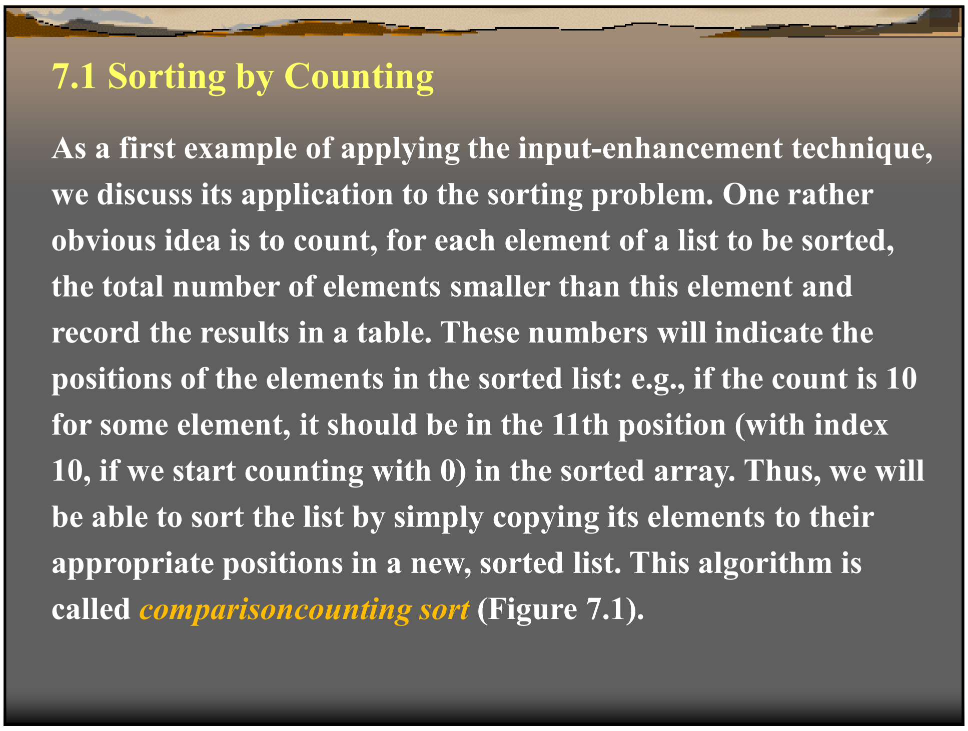


The other type of technique that exploits space-for-time trade-offs simply uses extra space to facilitate faster and/or more flexible access to the data.We call this approach *prestructuring*. This name highlights two facets of this variation of the space-for-time trade-off: some processing is done before a problem in Question is actually solved but, unlike the input-enhancement variety, it deals with access structuring. We illustrate this approach by:

- hashing (Section 7.3)
- indexing with B-trees (Section 7.4)

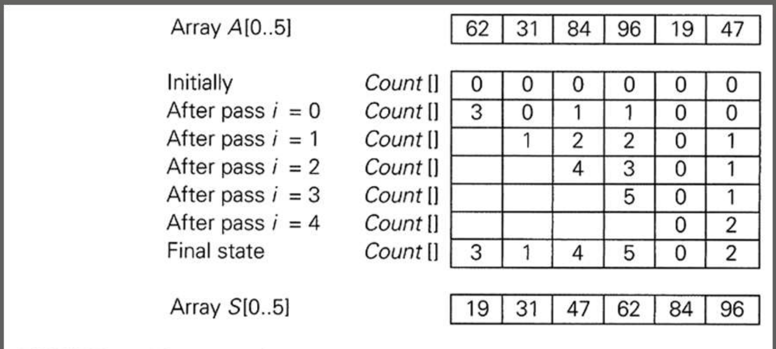


There is one more algorithm design technique related to the space-for-time trade-off idea: *dynamic programming*. This strategy is based on recording solutions to overlapping subproblems of a given problem in a table from which a solution to the problem in question is then obtained. We discuss this well-developed technique separately, in the next chapter of the book.



7.1 Sorting by Counting

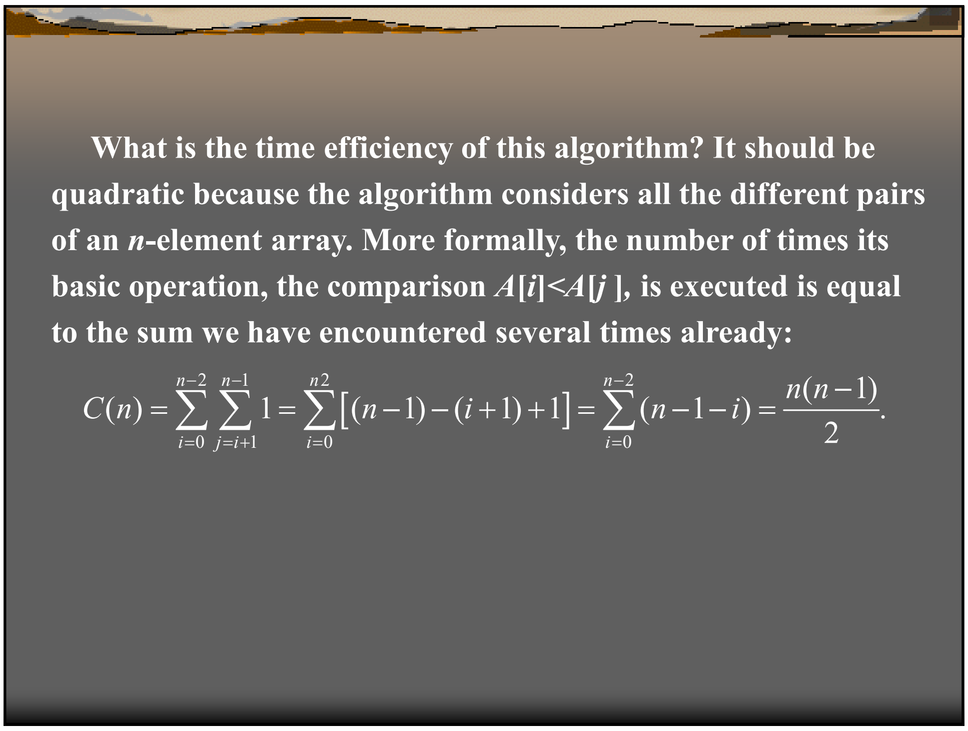
As a first example of applying the input-enhancement technique, we discuss its application to the sorting problem. One rather obvious idea is to count, for each element of a list to be sorted, the total number of elements smaller than this element and record the results in a table. These numbers will indicate the positions of the elements in the sorted list: e.g., if the count is 10 for some element, it should be in the 11th position (with index 10, if we start counting with 0) in the sorted array. Thus, we will be able to sort the list by simply copying its elements to their appropriate positions in a new, sorted list. This algorithm is called *comparisoncounting sort* (Figure 7.1).



Array *A*[0..5]		62	31	84	96	19	47
Initially	*Count* []	0	0	0	0	0	0
After pass *i* = 0	*Count* []	3	0	1	1	0	0
After pass *i* = 1	*Count* []		1	2	2	0	1
After pass *i* = 2	*Count* []			4	3	0	1
After pass *i* = 3	*Count* []				5	0	1
After pass *i* = 4	*Count* []					0	2
Final state	*Count* []	3	1	4	5	0	2
Array *S*[0..5]		19	31	47	62	84	96

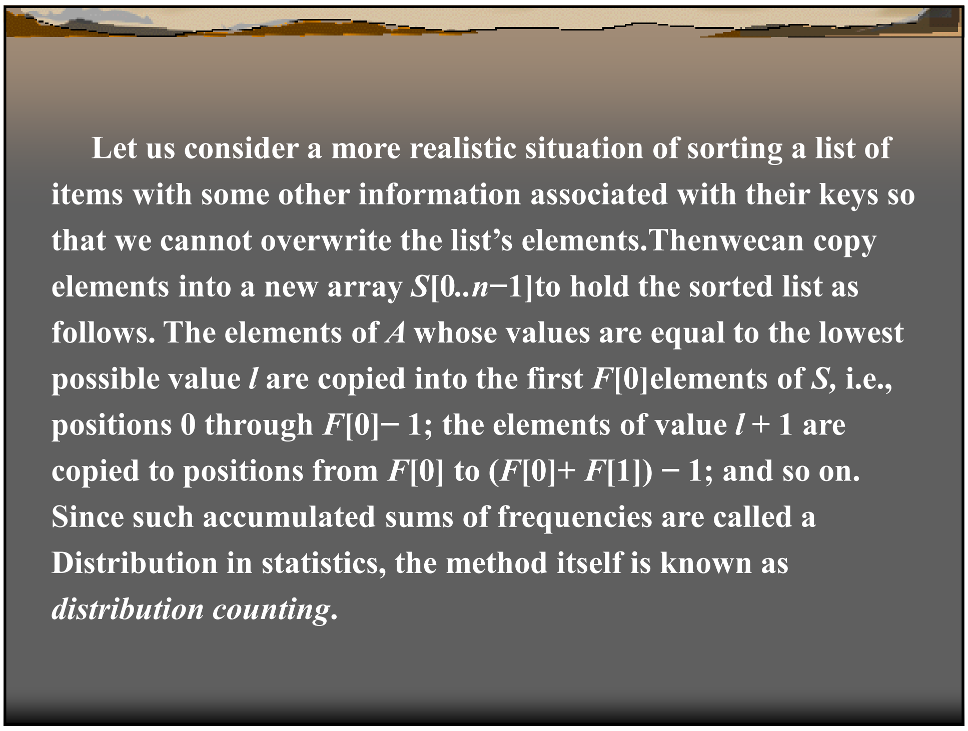
FIGURE 7.1 Example of sorting by comparison counting.

```
ALGORITHM    ComparisonCountingSort($A[0..n-1]$)
    //Sorts an array by comparison counting
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $S[0..n-1]$ of $A$'s elements sorted in
    nondecreasing order
    for $i \leftarrow 0$ to $n-1$ do $Count[i] \leftarrow 0$
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] < A[j]$
                $Count[j] \leftarrow Count[j] + 1$
            else $Count[i] \leftarrow Count[i] + 1$
    for $i \leftarrow 0$ to $n-1$ do $S[Count[i]] \leftarrow A[i]$
    return $S$
```

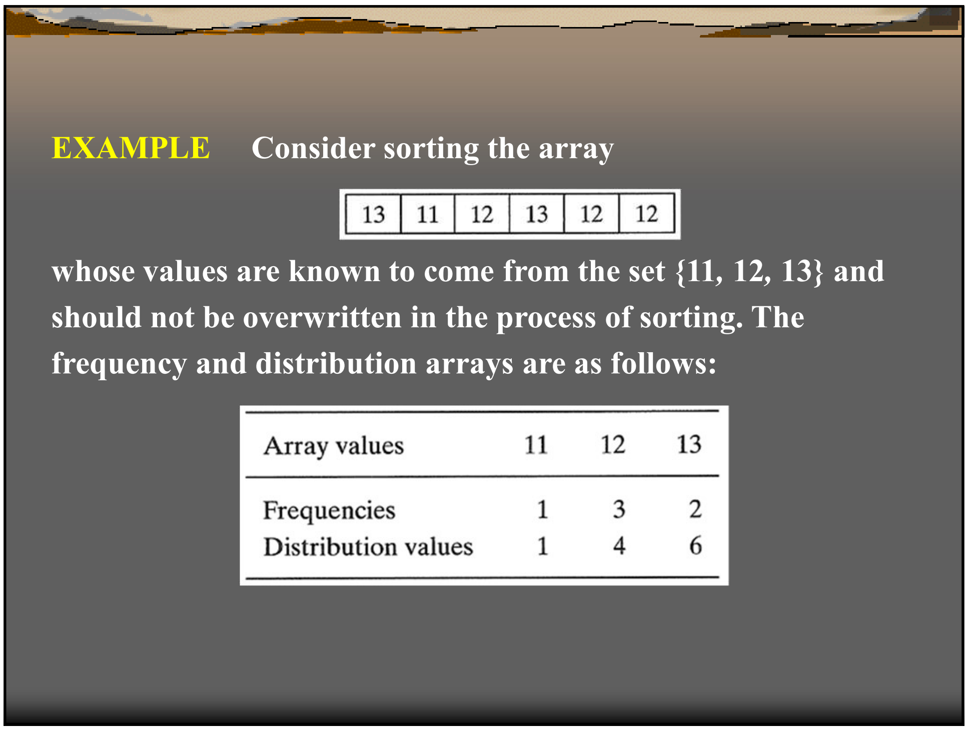


What is the time efficiency of this algorithm? It should be quadratic because the algorithm considers all the different pairs of an *n*-element array. More formally, the number of times its basic operation, the comparison $A[i]<A[j\ ]$, is executed is equal to the sum we have encountered several times already:

$$C(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n2}\left[(n-1)-(i+1)+1\right]=\sum_{i=0}^{n-2}(n-1-i)=\frac{n(n-1)}{2}.$$



Let us consider a more realistic situation of sorting a list of items with some other information associated with their keys so that we cannot overwrite the list's elements.Thenwecan copy elements into a new array $S[0..n-1]$to hold the sorted list as follows. The elements of A whose values are equal to the lowest possible value l are copied into the first $F[0]$elements of S, i.e., positions 0 through $F[0]-1$; the elements of value $l+1$ are copied to positions from $F[0]$ to $(F[0]+F[1])-1$; and so on. Since such accumulated sums of frequencies are called a Distribution in statistics, the method itself is known as *distribution counting*.

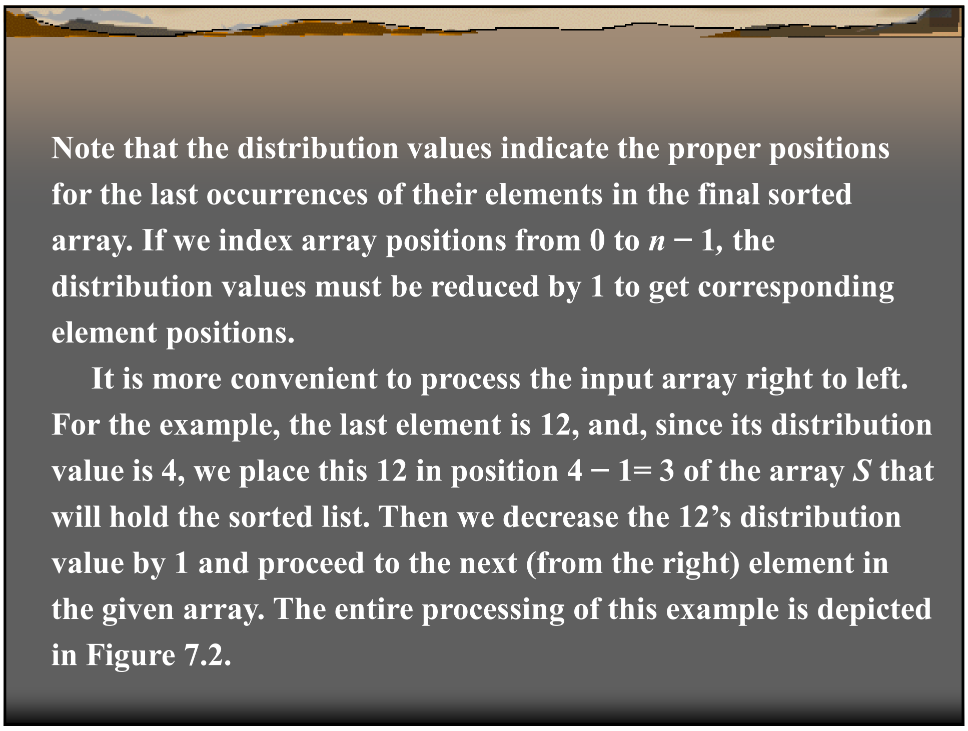


EXAMPLE Consider sorting the array

13	11	12	13	12	12

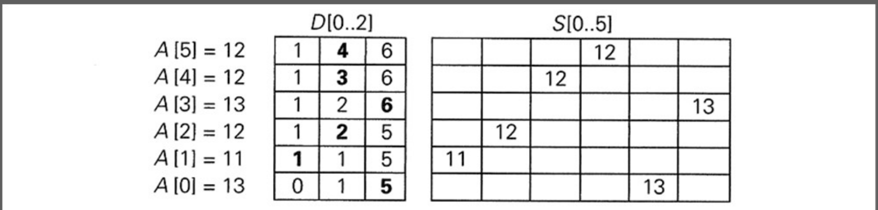
whose values are known to come from the set {11, 12, 13} and should not be overwritten in the process of sorting. The frequency and distribution arrays are as follows:

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6



Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n - 1$, the distribution values must be reduced by 1 to get corresponding element positions.

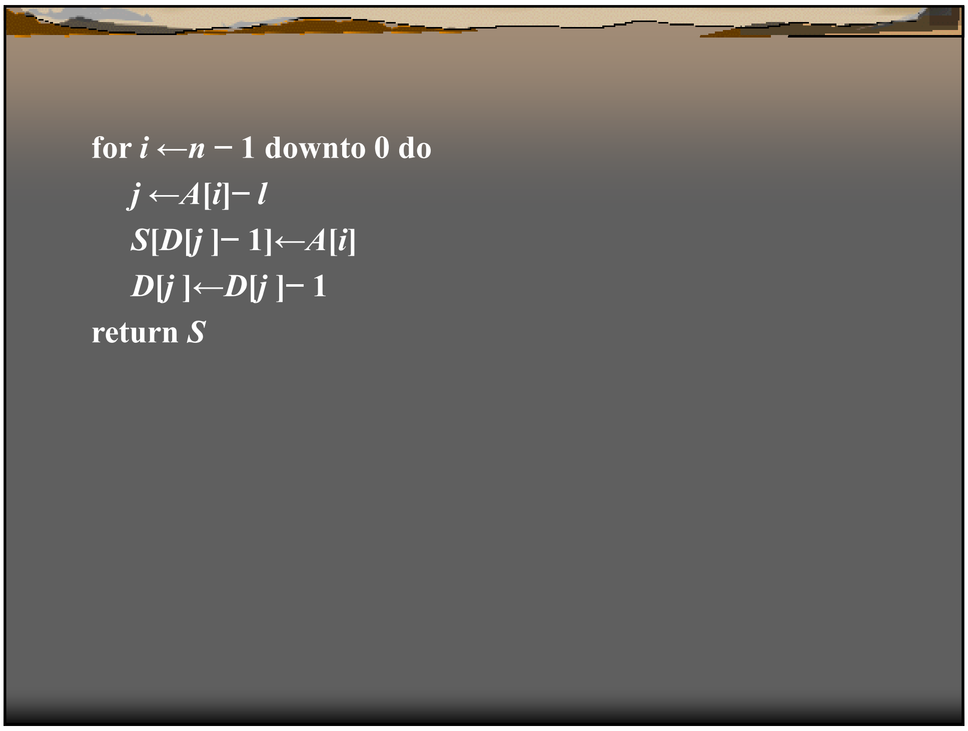
It is more convenient to process the input array right to left. For the example, the last element is 12, and, since its distribution value is 4, we place this 12 in position $4 - 1 = 3$ of the array S that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to the next (from the right) element in the given array. The entire processing of this example is depicted in Figure 7.2.



	D[0..2]			S[0..5]					
$A[5] = 12$	1	**4**	6				12		
$A[4] = 12$	1	**3**	6			12			
$A[3] = 13$	1	2	**6**						13
$A[2] = 12$	1	**2**	5		12				
$A[1] = 11$	**1**	1	5	11					
$A[0] = 13$	0	1	**5**					13	

FIGURE 7.2 Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

```
ALGORITHM    DistributionCountingSort(A[0..n − 1], l, u)
    //Sorts an array of integers from a limited range by
    distribution counting
    //Input: An array A[0..n − 1] of integers between l and u
    (l ≤ u)
    //Output: Array S[0..n − 1] of A's elements sorted in
    nondecreasing order
    for j ←0 to u − l do D[j ]←0                //initialize frequencies
    for i ←0 to n − 1 do D[A[i]− l]←D[A[i]− l]+ 1
                                                //compute frequencies
    for j ←1 to u − l do D[j ]←D[j − 1]+ D[j ]
                                                //reuse for distribution
```



```
for $i \leftarrow n - 1$ downto 0 do
    $j \leftarrow A[i] - l$
    $S[D[j] - 1] \leftarrow A[i]$
    $D[j] \leftarrow D[j] - 1$
return $S$
```