Introduction to

3rd Edition

The Design and Analysis of Algorithms

4

Decrease-and -Conquer

The *decrease-and-conquer* (減少和征服) technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance. Once such a relationship is established, it can be exploited either top down or bottom up.

(*decrease-and-conquer*是基於利用給定問題實例的解決方案及其較小實例的解決方案之間的關係。 一旦有了這樣的關係建立後，可以自上而下或自下而上地應用它。)

The former leads naturally to a recursive implementation, although, as one can see from several examples in this chapter, an ultimate implementation may well be nonrecursive. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the *incremental approach*.
(前者自然會導致遞迴實現，儘管從本章的幾個示例中可以看出，最終實現很可能是非遞迴的。 自下而上的變化通常是迭代實現的，從解決問題的最小實例開始； 它有時被稱為增量方法。)

There are three major variations of decrease-and-conquer:

- decrease by a constant
- decrease by a constant factor
- variable size decrease

In the *decrease-by-a-constant* variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

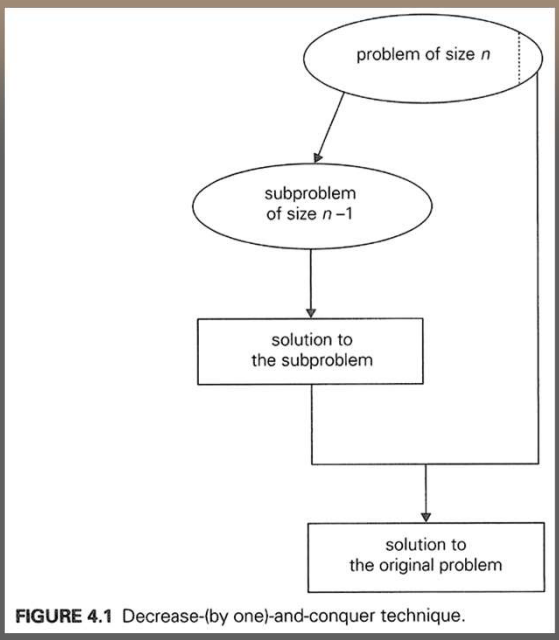

FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Consider, as an example, the exponentiation problem of computing a^n where $a \neq 0$ and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size $n - 1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1 & \text{if } n = 0, \end{cases} \qquad \textbf{(4.1)}$$

The *decrease-by-a-constant-factor* technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two. (Can you give an example of Such an algorithm?) The decrease-by-half idea is illustrated in Figure 4.2.

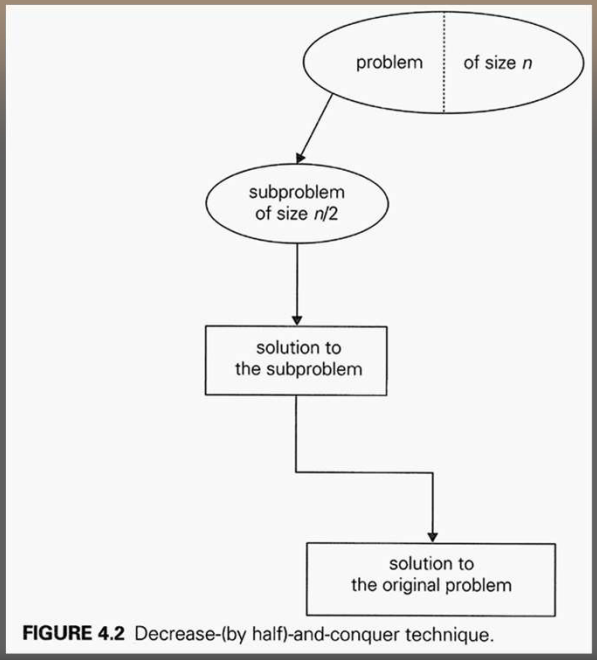

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive.} \\ (a^{n/2})^2 \cdot a & \text{if } n \text{ is odd.} \\ 1 & \text{if } n = 0. \end{cases}$$

(4.2)

Consider, as an example, the exponentiation problem of computing a^n where a 0 and n is a nonnegative integer. The relationship between a solution to an instance of size n and an instance of size $n - 1$ is obtained by the obvious formula $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive.} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd.} \\ 1 & \text{if } n = 0. \end{cases}$$

(4.1)

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances with integer exponents only, the former does not work for odd n. If n is odd, we have to compute a^{n-1} by using the rule for even-valued exponents and then multiply the result by a. To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive.} \\ (a^{n/2})^2 \cdot a & \text{if } n \text{ is odd.} \\ 1 & \text{if } n = 0. \end{cases} \qquad (4.2)$$

Finally, in the *variable-size-decrease* variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation. Recall that this algorithm is based on the formula

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

4.1 Insertion Sort

In this section, we consider an application of the decrease-by-one technique to sorting an array $A[0..n-1]$. Following the technique's idea, we assume that the smaller problem of sorting the array $A[0..n-2]$ has already been solved to give us a sorted array of size $n-1$: $A[0] \leq \cdots \leq A[n-2]$. How can we take advantage of this solution to the smaller problem to get a solution to the original problem by taking into account the element $A[n-1]$?

Obviously, all we need is to find an appropriate position for $A[n-1]$ among the sorted elements and insert it there. This is usually done by scanning the sorted subarray from right to left until the first element smaller than or equal to $A[n-1]$ is encountered to insert $A[n-1]$ right after that element. The resulting algorithm is called *straight insertion sort* or simply *insertion sort*.

ALGORITHM *InsertionSort*$(A[0..n-1])$

//Sorts a given array by insertion sort

//Input: An array $A[0..n-1]$ of n orderable elements

//Output: Array $A[0..n-1]$ sorted in nondecreasing order

for $i \leftarrow 1$ to $n-1$ do

  $v \leftarrow A[i]$

  $j \leftarrow i-1$

  while $j \geq 0$ and $A[j] > v$ do

    $A[j+1] \leftarrow A[j]$

    $j \leftarrow j-1$

  $A[j+1] \leftarrow v$

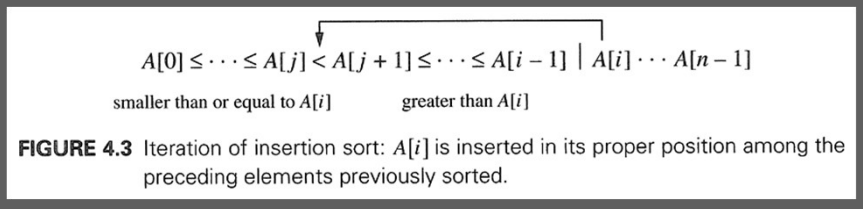

FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

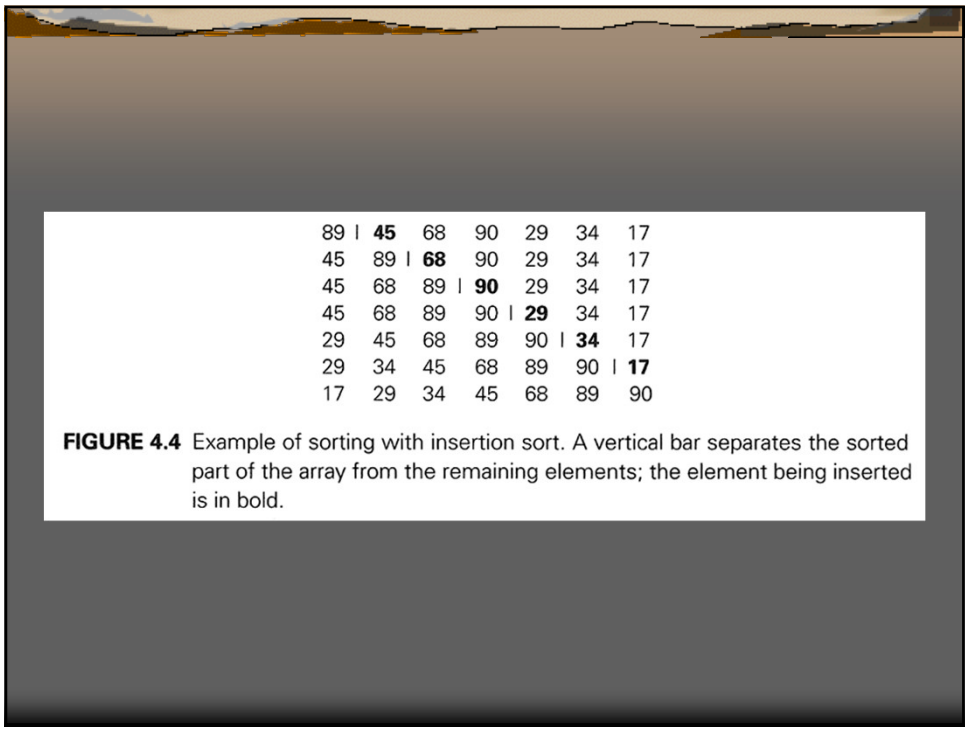

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$. (Note that we are using the fact that on the ith iteration of insertion sort all the elements preceding $A[i]$ are the first i elements in the input, albeit in the sorted order.) Thus, for the worst-case input, we get $A[0] > A[1]$ (for $i = 1$), $A[1] > A[2]$ (for $i = 2$), $\ldots$, $A[n-2] > A[n-1]$ (for $i = n - 1$). In other words, the worst-case input is an array of strictly decreasing values.

The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \le A[i]$ for every $i = 1, \ldots, n-1$, i.e., if the input array is already sorted in nondecreasing order. (Though it "makes sense" that the best case of an algorithm happens when the problem is already solved, it is not always the case, as you are going to see in our discussion of quicksort in Chapter 5.) Thus, for sorted arrays, the number of key comparisons is

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

A rigorous analysis of the algorithm's average-case efficiency is based on investigating the number of element pairs that are out of order (see Problem 11 in this section's exercises). It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$