

This suggests simply computing a^n by multiplying 1 by a n times.

We have already encountered at least two brute-force algorithms in the book: the consecutive integer checking algorithm for computing $\text{gcd}(m, n)$ in Section 1.1 and the definition-based algorithm for matrix multiplication in Section 2.3. Many other examples are given later in this chapter. (Can you identify a few algorithms you already know as being based on the brute-force approach?)

Though rarely a source of clever or efficient algorithms, the brute-force approach should not be overlooked as an important algorithm design strategy. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only general approach for which it is more difficult to point out problems it *cannot* tackle. Second, for some important problems—e.g., sorting, searching, matrix multiplication, string matching—the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed. Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. Finally, a brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

3.1 Selection Sort and Bubble Sort

In this section, we consider the application of the brute-force approach to the problem of sorting: given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order. As we mentioned in Section 1.3, dozens of algorithms have been developed for solving this very important problem. You might have learned several of them in the past. If you have, try to forget them for the time being and look at the problem afresh.

Now, after your mind is unburdened of previous knowledge of sorting algorithms, ask yourself a question: “What would be the most straightforward method for solving the sorting problem?” Reasonable people may disagree on the answer to this question. The two algorithms discussed here—selection sort and bubble sort—seem to be the two prime candidates.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last $n - 1$ elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the

i th pass through the list, which we number from 0 to $n - 2$, the algorithm searches for the smallest item among the last $n - i$ elements and swaps it with A_i :

$$A_0 \leq A_1 \leq \cdots \leq A_{i-1} \mid A_i, \dots, A_{\min}, \dots, A_{n-1}$$

in their final positions the last $n - i$ elements

After $n - 1$ passes, the list is sorted.

Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

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

```
//Sorts a given array by selection sort
//Input: An array  $A[0..n - 1]$  of orderable elements
//Output: Array  $A[0..n - 1]$  sorted in nondecreasing order
for  $i \leftarrow 0$  to  $n - 2$  do
     $\min \leftarrow i$ 
    for  $j \leftarrow i + 1$  to  $n - 1$  do
        if  $A[j] < A[\min]$   $\min \leftarrow j$ 
    swap  $A[i]$  and  $A[\min]$ 
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Figure 3.1.

The analysis of selection sort is straightforward. The input size is given by the number of elements n ; the basic operation is the key comparison $A[j] < A[\min]$. The number of times it is executed depends only on the array size and is given by the following sum:

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

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Since we have already encountered the last sum in analyzing the algorithm of Example 2 in Section 2.3, you should be able to compute it now on your own. Whether you compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, the answer, of course, must be the same:

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

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $\Theta(n)$, or, more precisely, $n-1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n-1$ passes the list is sorted. Pass i ($0 \leq i \leq n-2$) of bubble sort can be represented by the following diagram:

$$A_0, \dots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \dots, A_{n-i-1} \mid A_{n-i} \leq \dots \leq A_{n-1}$$

in their final positions

Here is pseudocode of this algorithm.

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

//Sorts a given array by bubble sort

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

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

for $i \leftarrow 0$ **to** $n-2$ **do**

for $j \leftarrow 0$ **to** $n-2-i$ **do**

if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Figure 3.2.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n ; it is obtained by a sum that is almost identical to the sum for selection sort:

89	$\overset{?}{\leftrightarrow}$	45		68		90		29		34		17
45		89	$\overset{?}{\leftrightarrow}$	68		90		29		34		17
45		68		89	$\overset{?}{\leftrightarrow}$	90	$\overset{?}{\leftrightarrow}$	29		34		17
45		68		89		29		90	$\overset{?}{\leftrightarrow}$	34		17
45		68		89		29		34		90	$\overset{?}{\leftrightarrow}$	17
45		68		89		29		34		17		90
45	$\overset{?}{\leftrightarrow}$	68	$\overset{?}{\leftrightarrow}$	89	$\overset{?}{\leftrightarrow}$	29		34		17		90
45		68		29		89	$\overset{?}{\leftrightarrow}$	34		17		90
45		68		29		34		89	$\overset{?}{\leftrightarrow}$	17		90
45		68		29		34		17		89		90
etc.												

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

$$\begin{aligned}
 C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] \\
 &= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).
 \end{aligned}$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

As is often the case with an application of the brute-force strategy, the first version of an algorithm obtained can often be improved upon with a modest amount of effort. Specifically, we can improve the crude version of bubble sort given above by exploiting the following observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm (Problem 12a in this section's exercises). Though the new version runs faster on some inputs, it is still in $\Theta(n^2)$ in the worst and average cases. In fact, even among elementary sorting methods, bubble sort is an inferior choice, and if it were not for its catchy name, you would probably have never heard of it. However, the general lesson you just learned is important and worth repeating:

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.