Introduction to

3rd Edition

The Design and Analysis of Algorithms

3

Brute Force and Exhaustive Search

The subject of this chapter is brute force (暴力法) and its important special case, exhaustive search (窮舉搜索法). Brute force can be described as follows:

> ***Brute force*** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

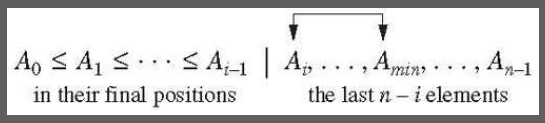
(暴力法是解決問題的直接方法，通常直接基於問題陳述和所涉及概念的定義。)

As an example, consider the exponentiation problem:

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}},$$

3.1 Selection Sort and Bubble Sort

Selection Sort (選擇排序法)



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 $A[min]$

| 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.

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:

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

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:

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

Bubble Sort (泡沫排序法)

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \cdots \leq A_{n-1}$$
in their final positions

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* ←0 to *n* − 2 do

  for *j* ←0 to *n* − 2 − *i* do

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

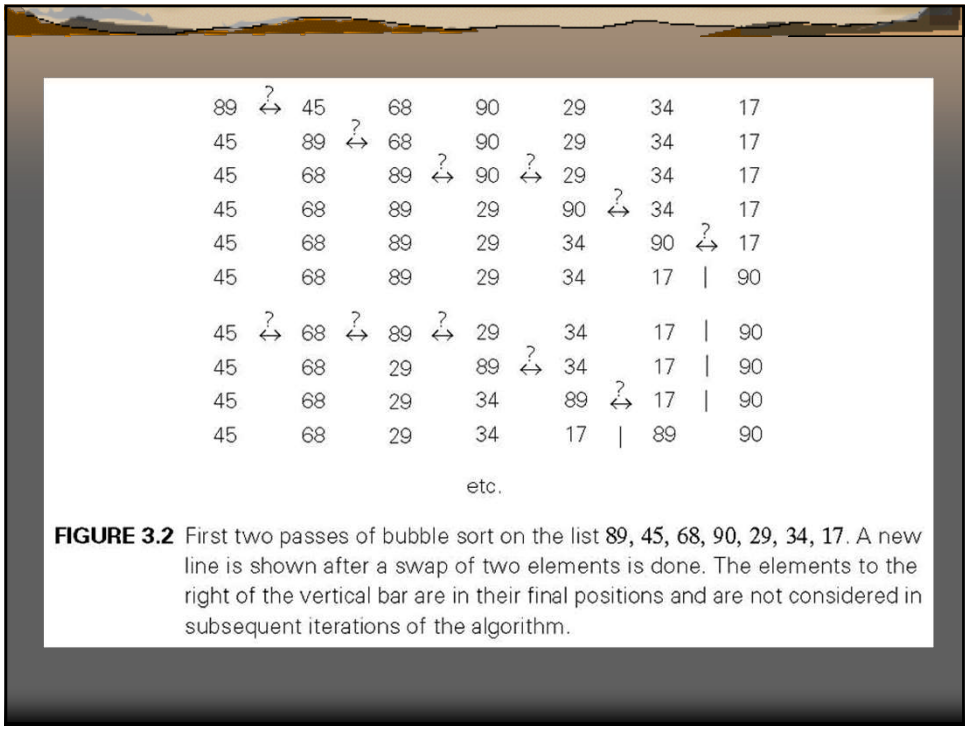


FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

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:

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

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

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:

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

3.2 Sequential Search and Brute-Force String Matching

Sequential Search

ALGORITHM *SequentialSearch2*(*A*[0..*n*], *K*)

//Implements sequential search with a search key as a sentinel (哨兵)

//Input: An array *A* of *n* elements and a search key *K*

//Output: The index of the first element in *A*[0..*n* − 1]

//whose value is equal to *K* or −1 if no such element is found

A[*n*]←*K*

i ←0 ≠

while *A*[*i*] *K* do

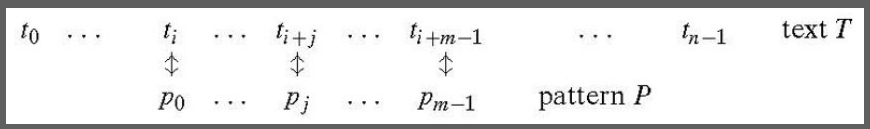
i ←*i* + 1

if *i* < *n* return *i*

else return −1

Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of *n* characters called the *text* and a string of *m* characters (*m* ≤ *n*) called the *pattern*, find a substring of the text that matches the pattern. To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching Substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_i+_{m-1} = p_{m-1}$:



ALGORITHM *BruteForceStringMatch*(*T* [0..*n* − 1], *P*[0..*m* − 1])

```
//Implements brute-force string matching
//Input: An array T [0..n − 1] of n characters representing a
  text and
// an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that
  starts a
// matching substring or −1 if the search is unsuccessful
for i ←0 to n − m do
    j ←0
    while j <m and P[j ]= T [i + j ] do
        j ←j + 1
    if j = m return i
return −1
```

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

3.3 Closest-Pair and Convex-Hull Problems by Brute Force

Closest-Pair Problem

For simplicity, we consider the two-dimensional case of the closest-pair problem. We assume that the points in question are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

ALGORITHM *BruteForceClosestPair*(P)

//Finds distance between two closest points in the plane by brute force

//Input: A list P of n $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$

//Output: The distance between the closest pair of points

$d \leftarrow \infty$

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

  for $j \leftarrow i + 1$ to n do

    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)2))$ //*sqrt* is square root

return d

Then the basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

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

$$= 2\left[(n-1) + (n-2) + \cdots + 1\right] = (n-1)n \in \Theta(n^2).$$