Variable-Size-Decrease Algorithms

In the third principal variety of decrease-and-conquer, the size reduction pattern varies from one iteration of the algorithm to another. Euclid's algorithm for computing the greatest common divisor (Section 1.1) provides a good example of this kind of algorithm. In this section, we encounter a few more examples of this variety.

Computing a Median and the Selection Problem

The selection problem is the problem of finding the kth smallest element in a list of n numbers. This number is called the kth order statistic. Of course, for $k = 1$ or $k = n$, we can simply scan the list in question to find the smallest or largest element, respectively. A more interesting case of this problem is for $k = \lceil n/2 \rceil$, which asks to find an element that is not larger than one half of the list's elements and not smaller than the other half. This middle value is called the ***median***, and it is one of the most important notions in mathematical statistics. Obviously, we can find the kth smallest element in a list by sorting the list first and then selecting the kth element in the output of a sorting algorithm. The time of such an algorithm is determined by the efficiency of the sorting algorithm used. Thus, with a fast sorting algorithm such as ***mergesort*** (discussed in the next chapter), the algorithm's efficiency is in $O(n \log n)$.
You should immediately suspect, however, that sorting the entire list is most likely overkill since the problem asks not to order the entire list but just to find its kth smallest element. Indeed, we can take advantage of the idea of ***partitioning*** a given list around some value p of, say, its first element. In general, this is a rearrangement of the list's elements so that the left part contains all the elements smaller than or equal to p, followed by the ***pivot*** p itself, followed by all the elements greater than or equal to p.

Of the two principal algorithmic alternatives to partition an array, here we discuss the **Lomuto partitioning**; we introduce the better known Hoare's algorithm in the next chapter. To get the idea behind the Lomuto partitioning, it is helpful to think of an array—or, more generally, a subarray $A[l..r]$ $(0 \le l \le r \le n-1)$—under consideration as composed of three contiguous segments. Listed in the order they follow pivot p, they are as follows: a segment with elements known to be smaller than p, the segment of elements known to be greater than or equal to p, and the segment of elements yet to be compared to p (see Figure 4.13a). Note that the segments can be empty; for example, it is always the case for the first two segments before the algorithm starts.
Starting with $i = l + 1$, the algorithm scans the subarray $A[l..r]$ left to right, maintaining this structure until a partition is achieved. On each iteration, it com- pares the first element in the unknown segment (pointed to by the scanning index i in Figure 4.13a) with the pivot p. If $A[i] \ge p$, i is simply incremented to expand the segment of the elements greater than or equal to p while shrinking the un- processed segment. If $A[i] < p$, it is the segment of the elements smaller than p that needs to be expanded. This is done by incrementing s, the index of the last element in the first segment, swapping $A[i]$ and $A[s]$, and then incrementing i to point to the new first element of the shrunk unprocessed segment. After no un-processed elements remain (Figure 4.13b), the algorithm swaps the pivot with $A[s]$ to achieve a partition being sought (Figure 4.13c).

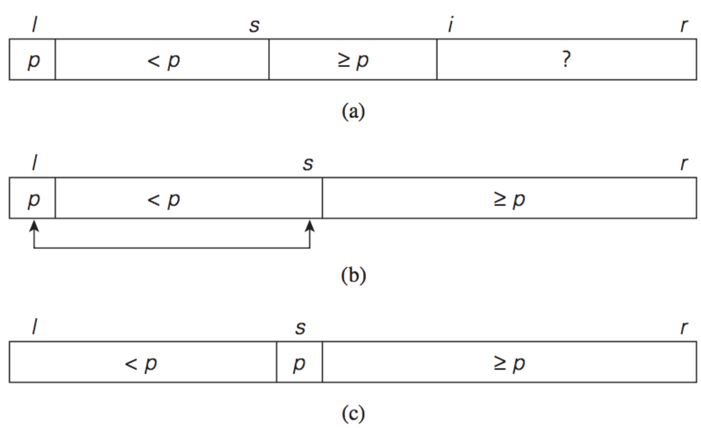

FIGURE 4.13 Illustration of the Lomuto partitioning.

Here is pseudocode implementing this partitioning procedure.

```
ALGORITHM  LomutoPartition(A[l..r])
    //Partitions subarray by Lomuto's algorithm using first element as pivot
    //Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
    //       indices l and r (l ≤ r)
    //Output: Partition of A[l..r] and the new position of the pivot
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p
            s ← s + 1;   swap(A[s], A[i])
    swap(A[l], A[s])
    return s
```

How can we take advantage of a list partition to find the kth smallest element in it? Let us assume that the list is implemented as an array whose elements are indexed starting with a 0, and let s be the partition's split position, i.e., the index of the array's element occupied by the pivot after partitioning. If $s = k - 1$, pivot p itself is obviously the kth smallest element, which solves the problem. If $s > k - 1$, the kth smallest element in the entire array can be found as the kth smallest element in the left part of the partitioned array. And if $s < k - 1$, it can be found as the $(k - s)$th smallest element in its right part. Thus, if we do not solve the problem outright, we reduce its instance to a smaller one, which can be solved by the same approach, i.e., recursively. This algorithm is called **quickselect**.

To find the kth smallest element in array $A[0..n - 1]$ by this algorithm, call *Quickselect*($A[0..n - 1]$, k) where

```
ALGORITHM  Quickselect(A[l..r], k)
    //Solves the selection problem by recursive partition-based algorithm
    //Input: Subarray A[l..r] of array A[0..n − 1] of orderable elements and
    //       integer k (1 ≤ k ≤ r − l + 1)
    //Output: The value of the kth smallest element in A[l..r]
    s ← LomutoPartition(A[l..r]) //or another partition algorithm
    if s = k − 1 return A[s]
    else if s > l + k − 1 Quickselect(A[l..s − 1], k)
    else Quickselect(A[s + 1..r], k − 1 − s)
```

In fact, the same idea can be implemented without recursion as well. For the non-recursive version, we need not even adjust the value of k but just continue until $s = k - 1$.

EXAMPLE Apply the partition-based algorithm to find the median of the following list of nine numbers: 4, 1, 10, 8, 7, 12, 9, 2, 15. Here, $k = \lceil 9/2 \rceil = 5$ and our task is to find the 5th smallest element in the array.

We use the above version of array partitioning, showing the pivots in bold.

0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*i*	
4	1	2	8	7	12	9	10	15
		s						*i*
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

Since $s = 2$ is smaller than $k - 1 = 4$, we proceed with the right part of the array:

0	1	2	3	4	5	6	7	8
			s	*i*				
			8	7	12	9	10	15
				s	*i*			
			8	7	12	9	10	15
				s				*i*
			8	7	12	9	10	15
			7	**8**	12	9	10	15

Now $s = k - 1 = 4$, and hence we can stop: the found median is 8, which is greater than 2, 1, 4, and 7 but smaller than 12, 9, 10, and 15. ■