**Computing a Median and the Selection Problem**

The ***selection problem*** is the problem of finding the *k*th smallest element in a list

of *n* numbers. This number is called the *k*th ***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.Amore interesting case of this problem is for *k* =\_*n/*2\_, 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 *k*th

smallest element in a list by sorting the list first and then selecting the *k*th 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

*k*th 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.*

p all are ≤p p all are ≥p

Of the two principal algorithmic alternatives to partition an array, here we

discuss the ***Lomuto partitioning*** [Ben00, p. 117]; 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 ≤ *l* ≤ *r* ≤ *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 compares

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*]≥ *p, i* is simply incremented to expand

the segment of the elements greater than or equal to *p* while shrinking the unprocessed

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

**4.5** Variable-Size-Decrease Algorithms **159**

(a)

(c)

*l s i r*

*p* < *p* ≥*p* ?

(b)

*l s r*

*p* < *p* ≥*p*

*l s r*

< *p p* ≥*p*

**FIGURE 4.13** Illustration of the Lomuto partitioning.

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 unprocessed

elements remain (Figure 4.13b), the algorithm swaps the pivot with*A*[*s*]

to achieve a partition being sought (Figure 4.13c).

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 *k*th 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 *k*th smallest element, which solves the problem. If

*s > k* − 1*,* the *k*th smallest element in the entire array can be found as the *k*th

smallest element in the left part of the partitioned array. And if *s < k* − 1*,* it can

**160** Decrease-and-Conquer

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 *k*th 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 *k*th 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

nonrecursive 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* = \_9*/*2\_ = 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:

**4.5** Variable-Size-Decrease Algorithms **161**

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.

How efficient is quickselect? Partitioning an *n*-element array always requires

*n* − 1 key comparisons. If it produces the split that solves the selection problem

without requiring more iterations, then for this best case we obtain *Cbest(n)* =

*n* − 1 ∈ *\_(n).* Unfortunately, the algorithm can produce an extremely unbalanced

partition of a given array, with one part being empty and the other containing *n* − 1

elements. In the worst case, this can happen on each of the *n* − 1 iterations. (For

a specific example of the worst-case input, consider, say, the case of *k* = *n* and a

strictly increasing array.) This implies that

*Cworst(n)* = *(n* − 1*)* + *(n* − 2*)* + *. . .* + 1= *(n* − 1*)n/*2 ∈ *\_(n*2*),*

which compares poorly with the straightforward sorting-based approach mentioned

in the beginning of our selection problem discussion.Thus, the usefulness of

the partition-based algorithm depends on the algorithm’s efficiency in the average

case. Fortunately, a careful mathematical analysis has shown that the average-case

efficiency is linear. In fact, computer scientists have discovered a more sophisticated

way of choosing a pivot in quickselect that guarantees linear time even in

the worst case [Blo73], but it is too complicated to be recommended for practical

applications.

It is also worth noting that the partition-based algorithm solves a somewhat

more general problem of identifying the *k* smallest and *n* − *k* largest elements of

a given list, not just the value of its *k*th smallest element.

The common/**vanilla quicksort** selects as a pivot the rightmost element. This has the consequence that it exhibits pathological performance O(N²) for a number of cases. In particular the sorted and the reverse sorted collections. In both cases the rightmost element is the worst possible element to select as a pivot. The pivot is ideally thought to me in the middle of the partitioning. The partitioning is supposed to split the data with the pivot into two sections, a low and a high section. Low section being lower than the pivot, the high section being higher.

**Median-of-three** pivot selection:

* select leftmost, middle and rightmost element
* order them to the left partition, pivot and right partition. Use the pivot in the same fashion as regular quicksort.

The common pathologies O(N²) of sorted / reverse sorted inputs are mitigated by this. *It is still easy to create pathological inputs to median-of-three. But it is a constructed and malicious use. Not a natural ordering.*

**ALGORITHM** *HoarePartition(A*[*l..r*]*)*

//Partitions a subarray by Hoare’s algorithm, using the first element

// as a pivot

//Input: Subarray of array *A*[0*..n* − 1]*,* defined by its left and right

// indices *l* and *r (l<r)*

//Output: Partition of *A*[*l..r*], with the split position returned as

// this function’s value

*p*←*A*[*l*]

*i* ←*l*; *j* ←*r* + 1

**repeat**

**repeat** *i* ←*i* + 1 **until** *A*[*i*]≥ *p*

**repeat** *j* ←*j* − 1 **until** *A*[*j* ]≤ *p*

swap*(A*[*i*]*, A*[*j* ]*)*

**until** *i* ≥ *j*

swap*(A*[*i*]*, A*[*j* ]*)* //undo last swap when *i* ≥ *j*

swap*(A*[*l*]*, A*[j])

return j;

**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*