4.4 Decrease-by-a-Constant-Factor algorithms

◆ **Binary Search**

Binary search **is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a** ***search key*** ***K*** **with the array's middle element** $A[m]$**. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if** $K < A[m]$**, and for the second half if** $K > A[m]$**:**

$$\underbrace{A[0] \ldots A[m-1]}_{\text{search here if } K<A[m]} \quad \overset{K}{\updownarrow} A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\text{search here if } K>A[m]}.$$

As an example, let us apply binary search to searching for $K = 70$ **in the array**

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//      or −1 if there is no such element
l←0; r ←n − 1
while l ≤ r do
    m← ⌊(l + r)/2⌋
    if K = A[m] then return m
    else if K <A[m] then r ←m − 1
    else l←m + 1
return −1
```

How many such comparisons does the algorithm make on an array of n elements? The answer obviously depends not only on n but also on the specifics of a particular instance of the problem. Let us find the number of key comparisons in the worst case $C_{worst}(n)$. The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}\left(\lfloor n/2 \rfloor\right) + 1 \quad \text{for } n > 1,\ C_{worst}(1) = 1. \qquad (4.3)$$

We already encountered recurrence (4.3), with a different initial condition, in Section 2.4 (see recurrence (2.4) and its solution there for $n = 2^k$). For the initial condition $C_{worst}(1) = 1$, we obtain

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1. \qquad (4.4)$$

Further, similarly to the case of recurrence (2.4) (Problem 7 in Exercises 2.4), the solution given by formula (4.4) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil. \qquad (4.5)$$

What can we say about the average-case efficiency of binary search? A sophisticated analysis shows that the average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \approx \log_2 n.$$

Fake-Coin Problem

We can easily set up a recurrence relation for the number of weighings $W(n)$ needed by this algorithm in the worst case:

$$W(n) = W(n/2) + 1 \quad \text{for } n > 1,\ W(1) = 0.$$

◆ Russian Peasant Multiplication

Now we consider a nonorthodox algorithm for multiplying two positive integers called *multiplication `a la russe* or the *Russian peasant method*. Let n and m be positive integers whose product we want to compute, and let us measure the instance size by the value of n. Now, if n is even, an instance of half the size has to deal with $n/2$, and we have an obvious formula relating the solution to the problem's larger instance to the solution to the smaller one:

$$n \cdot m = \frac{n}{2} \cdot 2m.$$

If *n* is odd, we need only a slight adjustment of this formula:

$$n \cdot m = \frac{n-1}{2} \cdot 2m + m.$$

n	*m*	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	*m*	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

FIGURE 4.11 Computing 50 · 65 by the Russian peasant method.

Josephus Problem

Our last example is the *Josephus problem*, named for Flavius Josephus, a famous Jewish historian who participated in and chronicled the Jewish revolt of 66–70 c.e. against the Romans. Josephus, as a general, managed to hold the fortress of Jotapata for 47 days, but after the fall of the city he took refuge with 40 diehards in a nearby cave.

(我們的最後一個例子是約瑟夫斯問題，以弗拉維烏斯約瑟夫斯的名字命名，弗拉維烏斯約瑟夫斯是一位著名的猶太歷史學家，他參與並記錄了公元 66-70 年的猶太人起義，反對羅馬人。 約瑟夫斯作為將軍，成功守住了約塔帕塔要塞47 天。但在城市淪陷後，他在附近的一個山洞裡與 40 名頑固分子一起避難。)

There, the rebels voted to perish rather than surrender. Josephus proposed that each man in turn should dispatch his neighbor, the order to be determined by casting lots. Josephus contrived to draw the last lot, and, as one of the two surviving men in the cave, he prevailed upon his intended victim to surrender to the Romans.

(在那裡，叛亂者投票選擇滅亡而不是投降。 約瑟夫斯提議每個人輪流殺死他的鄰居，順序由抽籤決定。 約瑟夫斯設法抽到最後一張，作為洞穴中兩個倖存的人之一，他說服了另一個倖存者向羅馬人投降。)

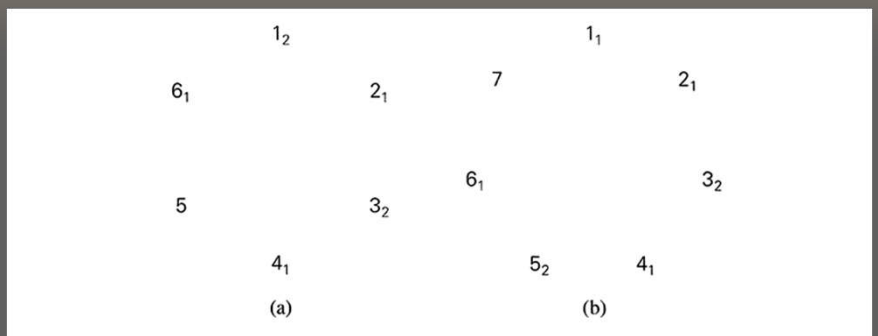

FIGURE 4.12 Instances of the Josephus problem for (a) $n = 6$ and (b) $n = 7$. Subscript numbers indicate the pass on which the person in that position is eliminated. The solutions are $J(6) = 5$ and $J(7) = 7$, respectively.

It is convenient to consider the cases of even and odd n's separately. If n is even, i.e., $n = 2k$, the first pass through the circle yields an instance of exactly the same problem but half its initial size. The only difference is in position numbering; for example, a person in initial position 3 will be in position 2 for the second pass, a person in initial position 5 will be in position 3, and so on (check Figure 4.12a). It is easy to see that to get the initial position of a person, we simply need to multiply his new position by 2 and subtract 1. This relationship will hold, in particular, for the survivor, i.e.,

$$J(2k) = 2J(k) - 1.$$

Let us now consider the case of an odd n ($n > 1$), i.e., $n = 2k + 1$. The first pass eliminates people in all even positions. If we add to this the elimination of the person in position 1 right after that, we are left with an instance of size k. Here, to get the initial position that corresponds to the new position numbering, we Have to multiply the new position number by 2 and add 1 (check Figure 4.12b). Thus, for odd values of n, we get

$$J(2k + 1) = 2J(k) + 1.$$

4.5 Variable-Size-Decrease Algorithms

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* (中位數), and it is one of the 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* (樞軸) *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* [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 $A[l..r]$ $(0 \le l \le r \le n - 1)$—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.

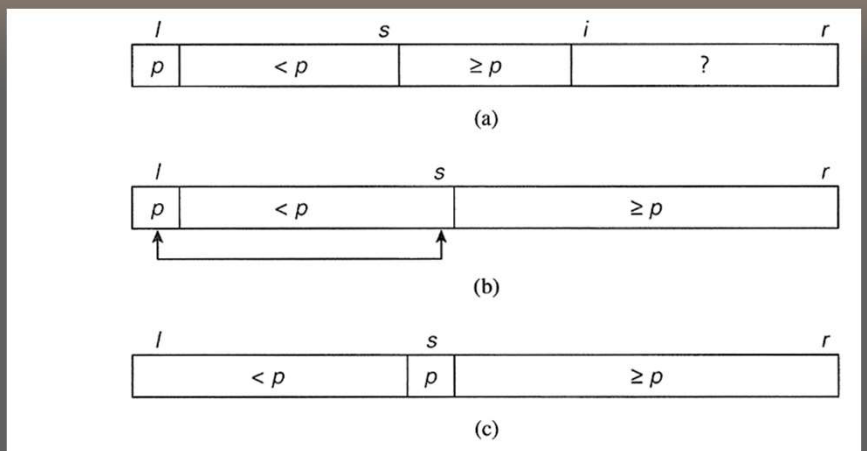

FIGURE 4.13 Illustration of the Lomuto partitioning.

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 $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ to r do
    if $A[i]<p$
        $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

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 $(1 \le k \le r - l + 1)$

//Output: The value of the kth smallest element in $A[l..r]$

$s \leftarrow$ *LomutoPartition*$(A[l..r])$ //or another partition algorithm

if $s = l + k - 1$ return $A[s]$

else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$

else *Quickselect*$(A[s+1..r], l + k - s - 1)$

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

$$l \leftarrow 0; r \leftarrow 8$$

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 $l+k-1 = 4$, we proceed with the right part of the array:

$k \leftarrow l+k-s-1=2;\ l \leftarrow s+1=3;\ r \leftarrow r=8;$

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 $s = l+k-1 = 4$, 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 $C_{best}(n) = n-1 \in \Theta(n)$. 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

$C_{worst}(n) = (n-1) + (n-2) + \ldots + 1 = (n-1)n/2 \in \Theta(n^2)$,