4.4 Decrease-by-a-Constant-Factor Algorithms

You may recall from the introduction to this chapter that decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. As an example of an algorithm based on this technique, we mentioned there exponentiation by squaring defined by formula (4.2). In this section, you will find a few other examples of such algorithms.. The most important and well-known of them is binary search. Decrease-by-a-constant-factor algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

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 $K > A[m]$:

$$\begin{array}{ccc} & K & \\ & \updownarrow & \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K<A[m]}} & A[m] & \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K>A[m]}}. \end{array}$$

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				

Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

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] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array. Moreover, for the sake of simplicity, we will count the so-called three-way comparisons. This assumes that after one comparison of K with $A[m]$, the algorithm can determine whether K is smaller, equal to, or larger than $A[m]$.

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 $C_{worst}(n)$:

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

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

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

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

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:

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

Formula (4.5) deserves attention. First, it implies that the worst-case time efficiency of binary search is in $\Theta(\log n)$. Second, it is the answer we should have

fully expected: since the algorithm simply reduces the size of the remaining array by about half on each iteration, the number of such iterations needed to reduce the initial size n to the final size 1 has to be about $\log_2 n$. Third, to reiterate the point made in Section 2.1, the logarithmic function grows so slowly that its values remain small even for very large values of n. In particular, according to formula (4.5), it will take no more than $\lceil \log_2(10^3 + 1) \rceil = 10$ three-way comparisons to find an element of a given value (or establish that there is no such element) in any sorted array of one thousand elements, and it will take no more than $\lceil \log_2(10^6 + 1) \rceil = 20$ comparisons to do it for any sorted array of size one million!

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:

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

(More accurate formulas for the average number of comparisons in a successful and an unsuccessful search are $C_{avg}^{yes}(n) \approx \log_2 n - 1$ and $C_{avg}^{no}(n) \approx \log_2(n + 1)$, respectively.)

Though binary search is an optimal searching algorithm if we restrict our operations only to comparisons between keys (see Section 11.2), there are searching algorithms (see interpolation search in Section 4.5 and hashing in Section 7.3) with a better average-case time efficiency, and one of them (hashing) does not even require the array to be sorted! These algorithms do require some special calculations in addition to key comparisons, however. Finally, the idea behind binary search has several applications beyond searching (see, e.g., [Ben00]). In addition, it can be applied to solving nonlinear equations in one unknown; we discuss this continuous analogue of binary search, called the method of bisection, in Section 12.4.

Fake-Coin Problem

Of several versions of the fake-coin identification problem, we consider here the one that best illustrates the decrease-by-a-constant-factor strategy. Among n identical-looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even, the balance scale will tell whether the sets weigh the same or which of the sets is heavier than the other but not by how much. The problem is to design an efficient algorithm for detecting the fake coin. An easier version of the problem—the one we discuss here—assumes that the fake coin is known to be, say, lighter than the genuine one.[1]

The most natural idea for solving this problem is to divide n coins into two piles of $\lfloor n/2 \rfloor$ coins each, leaving one extra coin aside if n is odd, and put the two

1. A much more challenging version assumes no additional information about the relative weights of the fake and genuine coins or even the presence of the fake coin among n given coins. We pursue this more difficult version in the exercises for Section 11.2.