Algorithms : From P to NP (Moret, Shapiro)

so that $t(n)$ is $O(n^{a \log n})$ for some suitable constant, a. Thus $t(n)$ grows no faster than a subexponential. In fact, $t(n)$ is $\Omega(n^{b \log n})$, as can be shown by further arguments in the same style.[4]

Average-Case Analysis: Binary Search Trees

We now turn to average-case analysis. As noted earlier, such analysis presupposes knowledge of the probability distribution of the instances of the problem; as we also noted, such knowledge is, in fact, rarely available, so that a uniform distribution (all instances are equally likely) is typically assumed. The difficulties associated with worst-case analysis all appear in average-case analysis, but they are compounded by the need to compute averages, i.e., expressions of the form $\sum_i p_i f(i)$, where p_i is the instance probability of object i. Such sums may not always lend themselves to reduction to a closed form and thus encumber the analysis throughout. As even uniform distributions often give rise to binomial coefficients, average-case analysis often requires familiarity with the manipulation of such coefficients.

Consider the problem of characterizing the average behavior of standard binary search trees. Although the worst-case behavior for all three operations (search, insertion, and deletion) is linear, as is easily shown on a tree constructed from a sorted list, it is well known that these trees behave much better in practice and usually exhibit logarithmic behavior. How can we prove that such is indeed the correct average behavior, say for insertion? We begin by postulating the usual assumption of uniformity: for a given input size n, all $n!$ distinct input sequences (of n insertions) are equally likely. Now let us build the binary search tree from the "average" input sequence, which we denote $k_1, k_2, \ldots, k_n$. The first key in the sequence becomes the root of the tree, thereby splitting our task into two subtasks: building the left subtree and building the right subtree, respectively. The left subtree contains all keys smaller than k_1; assume that there are n_l such keys and let $n_r = n - 1 - n_l$. Note that the $(n-1)!$ possible input sequences beginning with key k_1 consist of all possible mergings of the $n_l!$ possible sequences of keys smaller than k_1 and the $n_r!$ possible sequences of keys larger than k_1. This property allows us to proceed recursively.

[4]Program 2.2 is a good example of a "reluctant" algorithm; in fact, this could be termed a "multiply-and-surrender" algorithm (as opposed to the divide-and-conquer algorithms of Chapter 7). In the words of the inventors of this algorithm: "The basic multiply and surrender strategy consists in replacing the problem at hand with two or more subproblems, each slightly simpler than the original, and continue multiplying subproblems and subsubproblems recursively in this fashion as long as possible. At some point the subproblems will all become so simple that their solution can no longer be postponed, and we will have to surrender. Experience shows that, in most cases, by the time this point is reached the total work will be substantially higher than what could have been wasted by a more direct approach."

Since each node, once inserted, remains in place, a suitable measure need only account for the distance from the root to every node in the final tree. One such measure is the internal path length, $I(T)$, which is simply the sum of these distances and which equals the total number of comparisons made by all the insertions while building the tree. The internal path length obeys the recurrence

$$I(T) = |T| - 1 + I(T_l) + I(T_r),$$

where T denotes a binary tree, $|T|$ its number of nodes, and T_l and T_r its left and right subtrees. Now we can write a recurrence for $I_{\text{av}}(n)$, the average internal path length of binary search trees over n keys:

$$I_{\text{av}}(n) = n - 1 + \frac{1}{n} \cdot \sum_{i=0}^{n-1} \big(I_{\text{av}}(i) + I_{\text{av}}(n-1-i)\big). \tag{2.14}$$

While the real base case is $I_{\text{av}}(1) = 0$, we use $I_{\text{av}}(0) = 0$, because we need a value to substitute into the recurrence. The sum includes all n possible choices for the root of the tree (i.e., all possible choices for k_1, the first key in the sequence); since each choice determines a unique partition of the keys into the left and the right subtrees, we simply use the defining recurrence for the internal path length to obtain (2.14).

Now, this recurrence is not in a form which we can handle, because it involves all terms of lower order. Such recurrences, called *full-history recurrences*, occur commonly in the analysis of algorithms and can almost always be reduced to a form with a fixed number of terms by the simple expedient of subtracting the value at $n-1$ from the value at n, with coefficients chosen so as to cancel lower-order terms. In the present case, we choose the subtraction

$$n \cdot I_{\text{av}}(n) - (n-1) \cdot I_{\text{av}}(n-1),$$

so that, upon substituting from Equation 2.14 and simplifying (the two sums cancel except for the highest term), we get

$$n \cdot I_{\text{av}}(n) - (n-1) \cdot I_{\text{av}}(n-1) = 2n - 2 + 2I_{\text{av}}(n-1)$$

or

$$n \cdot I_{\text{av}}(n) - (n+1) \cdot I_{\text{av}}(n-1) = 2n - 2.$$

Only two function terms appear; however, they do not have constant coefficients. This difficulty can be overcome by dividing throughout by $n(n+1)$ to yield

$$\frac{1}{n+1} \cdot I_{\text{av}}(n) - \frac{1}{n} \cdot I_{\text{av}}(n-1) = 2 \cdot \frac{n-1}{n(n+1)},$$

and by substituting $g(n) = I_{\text{av}}(n)/(n+1)$, to get the linear recurrence with constant coefficients

$$g(n) - g(n-1) = 2\cdot\frac{n-1}{n(n+1)}, \text{ with } g(0) = 0.$$

Repeated substitution gives

$$g(n) = 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)}.$$

But note that, for $i > 3$, we have

$$\frac{1}{i+3} < \frac{i-1}{i(i+1)} < \frac{1}{i+2},$$

so that we may write

$$g(n) = \Theta(\sum_{i=1}^{n}\frac{1}{i}).$$

The sum term appearing in this equation is known as a *harmonic number*, more precisely in this case, the nth harmonic number, H_n. Recalling from calculus that $\sum_{i=1}^{n} 1/i$ is bounded below by $\int_1^{n+1}(1/x)\,dx$ and bounded above by $1 + \int_1^n (1/x)\,dx$, we get

$$\ln(n+1) \le H_n \le \ln n + 1,$$

and thus $H_n = \Theta(\log n)$. Hence we have $g(n) = \Theta(\log n)$ and thus $I_{\text{av}}(n) = \Theta(n \log n)$. At great expense of time and patience, we could obtain a more precise characterization by keeping the driving term intact, but we have argued that algorithmic analysis should be in asymptotic terms and thus have no need for additional precision. The result confirms our expectations: the average internal path length of binary search trees is optimal in $\Theta(\ )$ terms.

Since a successful search stops at a node in the tree, its average behavior can be characterized in terms of the internal path length of the tree, namely, by $I(T)/|T|$. Since insertion takes place at external nodes, it corresponds to an unsuccessful search, and thus its behavior can be characterized in terms of the external path length of the tree, namely, by $E(T)/(|T|+1)$. A simple induction argument shows that $E(T) = I(T) + 2|T|$. Therefore, our results also imply that, on average, both successful and unsuccessful searches run in logarithmic time; however, the same reasoning cannot be extended to the average height of the trees—although it is, in fact, logarithmic as well.