

## Specifying Recurrences and Solving Them via Iteration

The running times of most inherently recursive procedures, such as MERGE-SORT, lend themselves to specification and analysis via *recurrences*. In this handout, we discuss how one specifies recurrences and how one solves such recurrences.

### 1 Specifying Recurrences

A recurrence is simply a mathematical formula which specifies the running time of the algorithm on  $n$  elements,  $T(n)$ , as a function of the running time on some smaller number of elements (e.g.,  $T(n/2)$ ) plus some amount of overhead. For example, consider MERGE-SORT. In order to MERGE-SORT  $n$  elements, one must

1. recursively call MERGE-SORT on the first and second halves of the  $n$  elements and then
2. merge the sorted subgroups returned by the recursive calls.

So, what is the running time of MERGE-SORT? If we let  $T(n)$  represent the total running time of MERGE-SORT on  $n$  elements, then in Step 1 above,  $T(n/2)$  must be the running time of MERGE-SORT on each of the first and second halves of those  $n$  elements. In Step 2, on the order of  $n$  operations is required to merge two sorted groups whose total size is  $n$ . Therefore, the total running time of MERGE-SORT,  $T(n)$ , is twice  $T(n/2)$  (for the recursive calls) plus on the order of  $n$  work to perform the merge. Pulling this all together, we have

$$T(n) = 2T(n/2) + n. \quad (1)$$

There are a few subtleties in the above description which we have glossed over:

1. What if  $n$  is not even so that  $n/2$  is not an integer? If  $n$  were 27, how could one recursively call MERGE-SORT on 13 1/2 elements?
2. What does “on the order of  $n$  operations” mean, anyway?
3. What happens when  $n$  gets small? The recursion eventually has to stop, right?

To answer the first question, we note that in reality MERGE-SORT splits the set of  $n$  elements as evenly as possible, and this is specified in the code for MERGE-SORT itself: one recursive call is on  $\lfloor n/2 \rfloor$  elements and the other recursive call is on  $\lceil n/2 \rceil$  elements. If  $n$  is even, we do have two recursive calls on exactly  $n/2$  elements each; if  $n$  is odd, we have recursive call on  $(n-1)/2$  and  $(n+1)/2$  elements, respectively. So in reality, our recurrence is more precisely

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n. \quad (2)$$

However, one can show that for the class of recurrences corresponding to the overwhelming majority of real recursive procedures and programs, the floors and ceilings do not affect the analysis in any significant way. In other words, Recurrence 2 “behaves” just like Recurrence 1.

To answer the second question, one must specify precisely what “effort” one is counting. The merge operation in MERGE-SORT, for example, involves comparing elements, copying elements, incrementing counters, etc., etc. Should we count compares? Compares and copies? Program lines executed? Microseconds of wall-clock time? The answer is to count all and none of them. The merge operation entails somewhere between  $\lfloor n/2 \rfloor$  and  $n$  compares and exactly  $n$  copies. Each compare or copy is associated with a few lines of program code, plus some constant overhead. Each line of code may be executed in some number of microseconds, etc. So, depending on what “effort” one is counting, we might assess the merge as requiring  $n$ ,  $2n$ ,  $15n + 6$ , or some similar amount of “effort.” However, each of these accountings is effectively the same, they’re just in different *units*:  $n$  compares,  $2n$  counts and compares,  $15n + 6$  program lines, and so on. It’s analogous to seconds vs. minutes vs. hours. What’s important is the fact that *all* of these accountings grow *linearly* in  $n$  (as opposed to, say,  $n^2$ ). In algorithmic analysis, one cares about the *asymptotic growth rate* (i.e., the function of  $n$ , say  $n$  vs.  $n^2$ ) and not constant factors, lower order terms, or specific “units” of accounting. Furthermore, one can show that for the class of recurrences corresponding to the overwhelming majority of real recursive procedures and programs, dropping the constant factors (e.g., the “15” in  $15n + 6$ ) and lower order terms (e.g., the “6” in  $15n + 6$ ) does not affect the asymptotic analysis in any way. Thus, while our recurrence may precisely be

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 15n + 6, \quad (3)$$

(as measured in units of program lines), Recurrence 3 “behaves” (asymptotically) just like Recurrence 1. Dropping constant factors and lower order terms yields what is referred to as “order notation;” for example, “ $15n + 6$ ” is said to be “on the order of  $n$ .” Order notation is used so often in mathematics and computer science that a special notation has been developed for it: one would write  $15n + 6 = \Theta(n)$ , where  $\Theta$  is the Greek capital letter “theta.” Variants on this notation include  $O$  (big-oh) and  $\Omega$  (big-omega) which roughly correspond to “at most on the order of” (big-oh) and “at least on the order of” (big-omega).  $\Theta$ ,  $O$ , and  $\Omega$  are the asymptotic analogues of  $=$ ,  $\leq$ , and  $\geq$ .

Finally, to answer the third question, we note that recursive procedures do eventually terminate when some base condition is met. In the case of MERGE-SORT, a one item list need not be recursive split in order to be sorted: it is already (trivially) sorted. Thus, MERGE-SORT returns in (on the order of) one unit of time when called on a list of length one. Thus,  $T(1) = 1$ . This is a *base case* of the recurrence. Our recurrence is therefore more precisely

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1. \end{cases} \quad (4)$$

A recurrence may have more than one base case (for instance, a recursive procedure may specify different termination actions when  $n = 1$ ,  $n = 2$ , and  $n = 3$ ). However, for all recurrences corresponding to the overwhelming majority of real recursive procedures and programs, the base case(s) take some small, constant amount of “effort” and are thus “on the order of” 1. So, a recurrence of the form given by Recurrence 1 is *implicitly* assumed to be of the form given by Recurrence 4, unless the base case(s) are given *explicitly*. So, after all those subtleties, we are back where we started:

$$T(n) = 2T(n/2) + n.$$

How can one solve such recurrences?

## 2 Solving Recurrences

Consider our recurrence  $T(n) = 2T(n/2) + n$ . In order to solve the recurrence, it is good practice to first rewrite the recurrence with the recursive component *last* and to use a generic parameter

not to be confused with  $n$ . We may think of the following equation as our general pattern, which holds for any value of  $\square$ .

$$T(\square) = \square + 2T(\square/2) \tag{5}$$

Since our pattern (Equation 5) is valid for any value of  $\square$ , we may use it to “iterate” the recurrence as follows.

$$\begin{aligned} T(n) &= n + 2T(n/2) \\ &= n + 2\left(n/2 + 2T(n/2^2)\right) \\ &= n + n + 2^2T(n/2^2) \end{aligned} \tag{6}$$

$$= 2n + 2^2T(n/2^2) \tag{7}$$

Always simplify the expression, eliminating parentheses and combining terms as in Equations 6 and 7, before expanding further. Continuing...

$$\begin{aligned} T(n) &= 2n + 2^2\left(n/2^2 + 2T(n/2^3)\right) \\ &= 2n + n + 2^3T(n/2^3) \\ &= 3n + 2^3T(n/2^3) \\ &= 3n + 2^3\left(n/2^3 + 2T(n/2^4)\right) \\ &= 3n + n + 2^4T(n/2^4) \\ &= 4n + 2^4T(n/2^4) \end{aligned}$$

Notice the pattern that has been developed:

$$T(n) = n + 2T(n/2) = 2n + 2^2T(n/2^2) = 3n + 2^3T(n/2^3) = 4n + 2^4T(n/2^4).$$

Thus, we expect that for any  $k$ , we would have

$$T(n) = k \cdot n + 2^k T(n/2^k).$$

Formally, one must *prove* that this pattern is, indeed, correct (and we shall do so at the end of this handout), but *assuming* that it is correct, we may continue as follows.

Given that  $T(n) = k \cdot n + 2^k T(n/2^k)$  for all  $k$ , we next choose a value of  $k$  which causes our recurrence to reach a known base case, e.g.,  $T(1)$ . For what value of  $k$  does  $n/2^k = 1$ ? We must solve for  $k$  in this equation...

$$\begin{aligned} n/2^k &= 1 \\ \Leftrightarrow n &= 2^k \\ \Leftrightarrow \log_2 n &= k \end{aligned}$$

Since  $n/2^k = 1$  when  $k = \log_2 n$ , and  $T(1) = 1$ , we have

$$\begin{aligned} T(n) &= k \cdot n + 2^k T(n/2^k) \\ &= \log_2(n) \cdot n + 2^{\log_2 n} T(1) \\ &= n \log_2 n + n \cdot 1 \\ &= n \log_2 n + n \end{aligned}$$

Dropping the lower order term, we have that  $T(n)$  is on the order of  $n \log_2 n$ , and we would write

$$T(n) = \Theta(n \log_2 n).$$

Thus, MERGE-SORT is not quite as fast as linear search, which is  $\Theta(n)$ , but it is faster than INSERTION-SORT, which is  $\Theta(n^2)$ .

To complete our analysis, we next prove that the pattern we used was indeed correct; our proof is by induction.

**Claim 1** For all  $k \geq 1$ ,  $T(n) = k \cdot n + 2^k T(n/2^k)$ .

**Proof:** The proof is by induction on  $k$ . The base case,  $k = 1$ , is trivially true since the resulting equation matches the original recurrence. For the inductive step, assume that the statement is true for  $k - 1$ ; i.e.,

$$T(n) = (k - 1) \cdot n + 2^{k-1} T(n/2^{k-1}).$$

Our task is then to show that the statement is true for  $k$ . This may be accomplished by starting with this *inductive hypothesis* and applying the definition of the recurrence, as follows.

$$\begin{aligned} T(n) &= (k - 1) \cdot n + 2^{k-1} T(n/2^{k-1}) \\ &= (k - 1) \cdot n + 2^{k-1} \left( n/2^{k-1} + 2 T(n/2^k) \right) \\ &= (k - 1) \cdot n + n + 2^k T(n/2^k) \\ &= k \cdot n + 2^k T(n/2^k) \end{aligned}$$

□