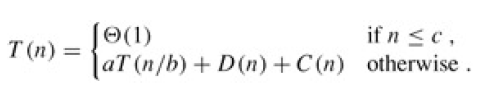
**Analyzing divide-and-conquer algorithms**

When an algorithm contains a recursive call to itself, its running time can often be described by a ***recurrence equation*** or ***recurrence***, which describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs. We can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

A recurrence for the running time of a divide-and-conquer algorithm is based on the three steps of the basic paradigm. As before, we let *T* (*n*) be the running time on a problem of size *n*. If the problem size is small enough, say *n* ≤ *c* for some constant *c*, the straightforward solution takes constant time, which we write as Θ(1). Suppose that our division of the problem yields *a* subproblems, each of which is 1/*b* the size of the original. (For merge sort, both *a* and *b* are 2, If we take *D*(*n*) time to divide the problem into subproblems and *C*(*n*) time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence



**Analysis of merge sort**

We set up the recurrence for *T* (*n*), the worst-case running time of merge sort on *n* numbers. Merge sort on just one element takes constant time. When we have *n* > 1 elements, we break down the running time as follows.

**Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, *D*(*n*) = Θ(1).

**Conquer:** We recursively solve two subproblems, each of size *n*/2, which contributes 2*T* (*n*/2) to the running time.

**Combine:** We have already noted that the MERGE procedure on an *n*-element subarray takes time Θ(*n*), so *C*(*n*) = Θ(*n*).

When we add the functions *D*(*n*) and *C*(*n*) for the merge sort analysis, we are adding a function that is Θ(*n*) and a function that is Θ(1). This sum is a linear function of *n*, that is, Θ(*n*). Adding it to the 2*T* (*n*/2) term from the "conquer" step gives the recurrence for the worst-case running time *T* (*n*) of merge sort:

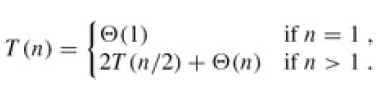
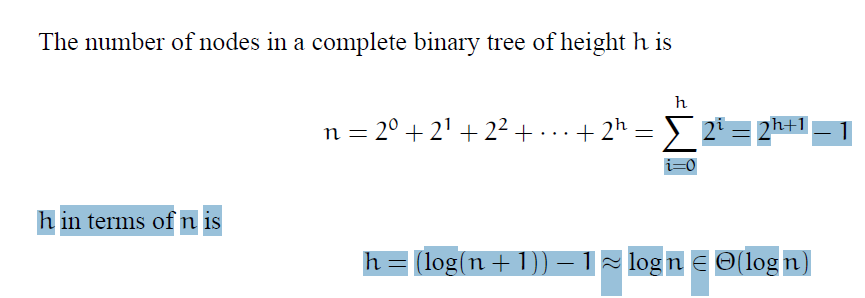
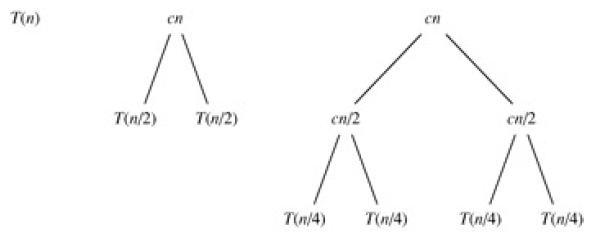


Figure coming next shows how we can solve the recurrence above. For convenience, we assume that *n* is an exact power of 2. Part (a) of the figure shows *T* (*n*), which in part (b) has been expanded into an equivalent tree representing the recurrence. The *cn* term is the root (the cost at the top level of recursion), and the two subtrees of the root are the two smaller recurrences *T* (*n*/2). Part (c) shows this process carried one step further by expanding *T* (*n*/2). The cost for each of the two subnodes at the second level of recursion is *cn*/2. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of *c*. Part (d) shows the resulting tree.





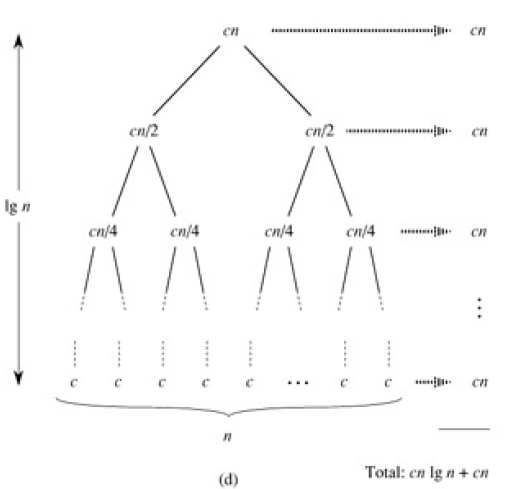


Figure 2.5: The construction of a recursion tree for the recurrence *T*(*n*) = 2*T*(*n*/2) + *cn*. Part

1. shows *T*(*n*), which is progressively expanded in *(b)-(d)* to form the recursion tree.

The fully expanded tree in part (d) has lg *n* + 1 levels (i.e., it has height lg *n*, as indicated), and each level contributes a total cost of *cn*. The total cost, therefore, is *cn* lg *n* + *cn*, which is Θ(*n*lg *n*).

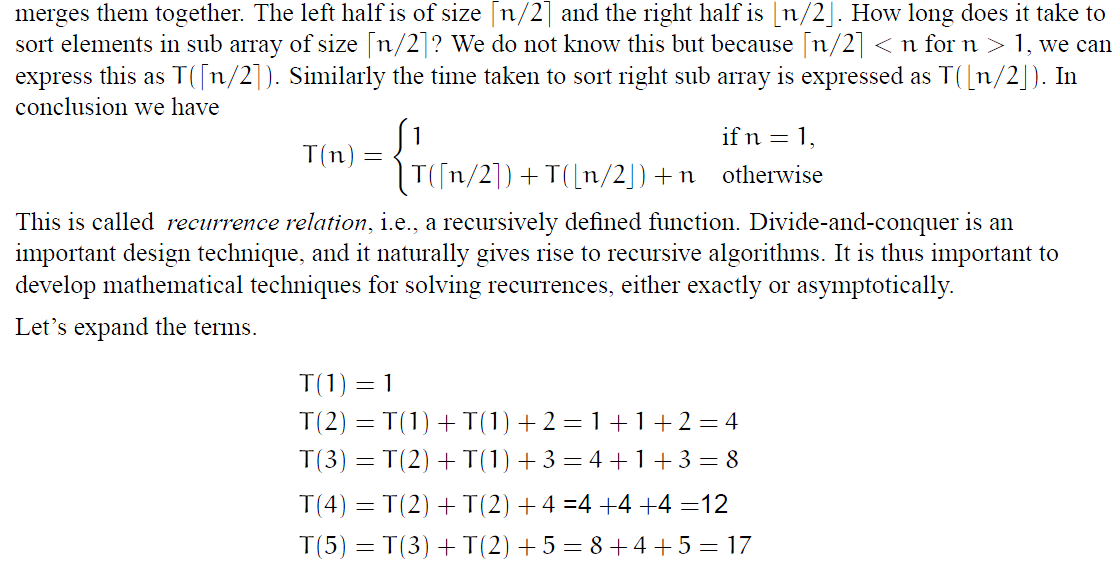
**Analysis of Merge Sort**

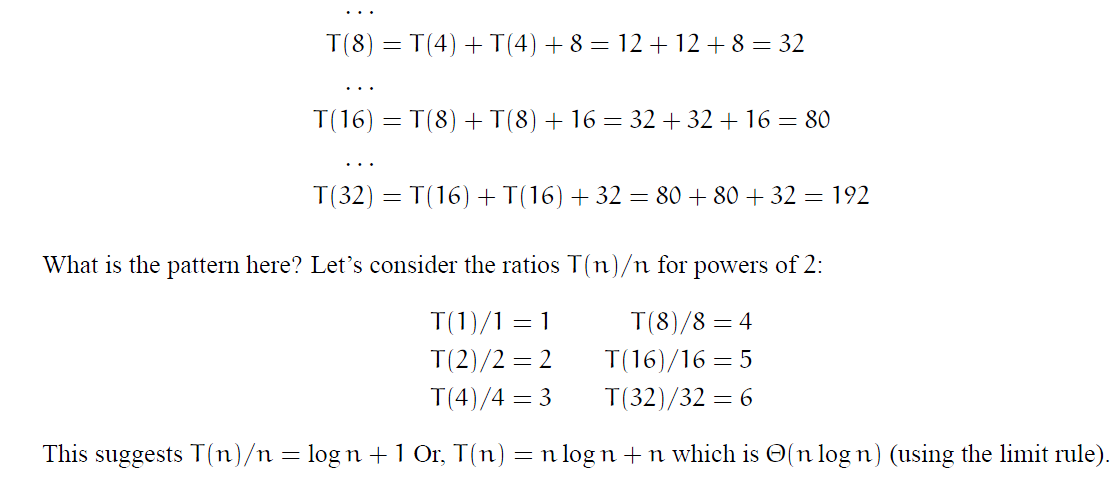
First let us consider the running time of the procedure Merge(A, p, q, r). Let n = r − p + 1 denote the total length of both the left and right sub-arrays. What is the running time of Merge as a function of n?

The algorithm contains three loops (none nested in the other). It is easy to see that each loop can be executed at most n times. (If you are a bit more careful you can actually see that all the loops together can only be executed n times in total, because each execution copies one new element to the array L and R, and L and R both have space for n elements) Thus the running time to Merge n items is Ꝋ(n). Let us write this without the asymptotic notation, simply as n.

Now, how do we describe the running time of the entire MergeSort algorithm? We will do this through the use of a recurrence, that is, a function that is defined recursively in terms of itself.

To avoid circularity, the recurrence for a given value of n is defined in terms of values that are strictly smaller than n. Finally, a recurrence has some basis values (e.g. for n = 1), which are defined explicitly.

Let T(n) denote the worst case running time of MergeSort on an array of length n. If we call MergeSort with an array containing a single item (n = 1) then the running time is constant. We can just write T(n) = 1, ignoring all constants. For n > 1, MergeSort splits into two halves, sorts the two and then 



n value in T(n)/n should be power of 2.

n=1,2,4,8,16

20=1, 21=2, 22=4

Time complexity of Merge Sort is nlogn in all 2 cases (worst and best) as merge sort always divides the array into two halves and take linear time to merge two halves.