Analysis of Merge Sort

Given that the merge function runs in Θ(n) time when merging n elements, how do we get to showing that mergeSort runs in Θ(nlgn) time? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we're sorting a total of n elements in the entire array.

The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint qq of the indices p and r. Recall that in big-Θ notation, we indicate constant time by Θ(1).

The conquer step, where we recursively sort two subarrays of approximately n/2 elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.

The combine step merges a total of n elements, taking Θ(n) time.

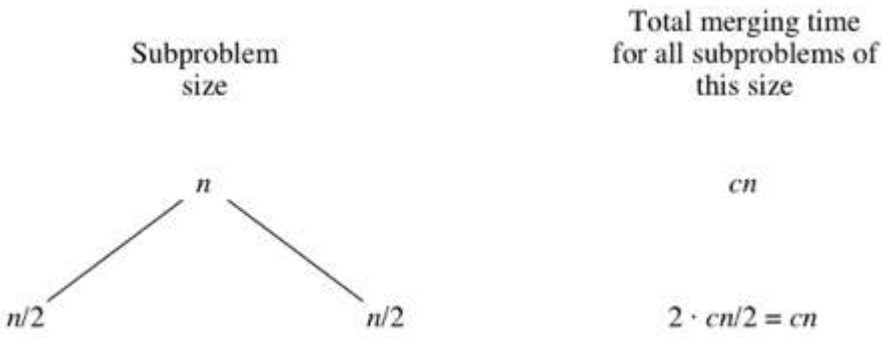
If we think about the divide and combine steps together, the Θ(1) running time for the divide step is a low-order term when compared with the Θ(n) running time of the combine step. So let's think of the divide and combine steps together as taking Θ(n) time. To make things more concrete, let's say that the divide and combine steps together take cn time for some constant c.

To keep things reasonably simple, let's assume that if n>1, then n is always even, so that when we need to think about n/2, it's an integer. (Accounting for the case in which n is odd doesn't change the result in terms of big-Θ notation.) So now we can think of the running time of mergeSort on an n-element subarray as being the sum of twice the running time of mergeSort on an (n/2)-element subarray (for the conquer step) plus cn (for the divide and combine steps—really for just the merging).

Now we have to figure out the running time of two recursive calls on n/2 elements. Each of these two recursive calls takes twice of the running time of

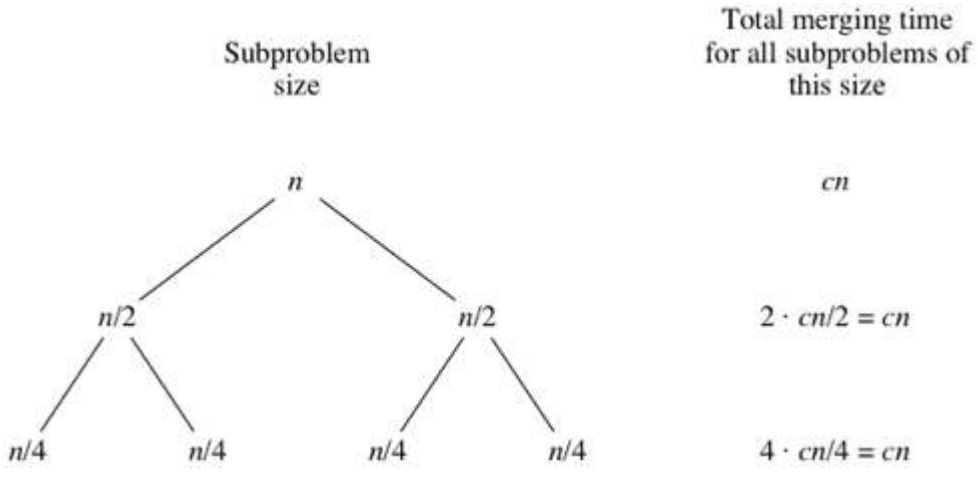
mergeSort on an (n/4)(n/4)-element subarray (because we have to halve n/2n/2) plus cn/2cn/2 to merge. We have two subproblems of size n/2, and each takes cn/2 time to merge, and so the total time we spend merging for subproblems of size n/2 is ***2·cn/2=cn***.

Let's draw out the merging times in a "tree":



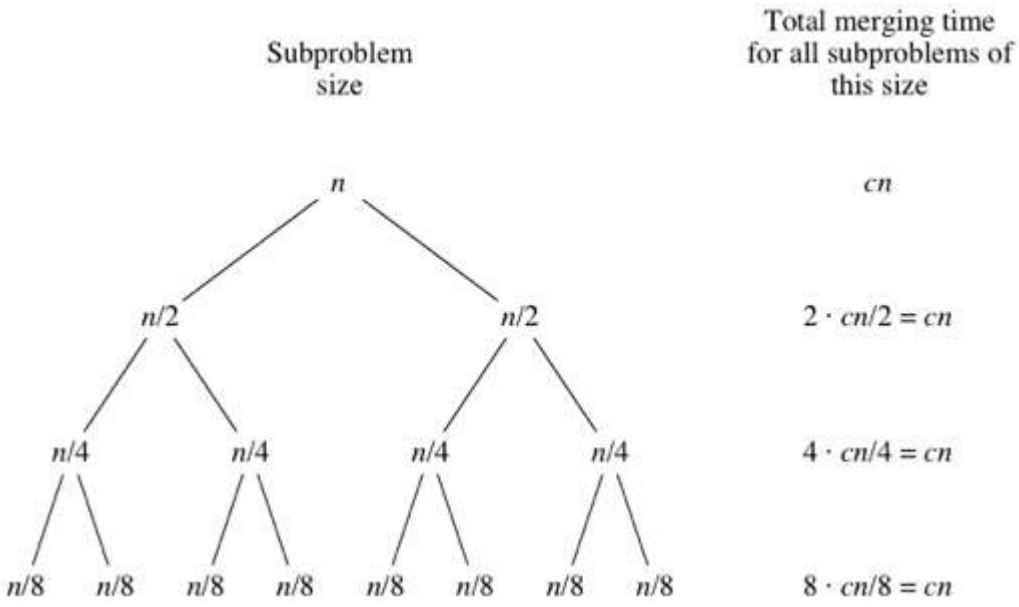
Computer scientists draw trees upside-down from how actual trees grow. A tree is a graph with no cycles (paths that start and end at the same place). Convention is to call the vertices in a tree its nodes. The root node is on top; here, the root is labeled with the n subarray size for the original n-element array. Below the root are two child nodes, each labeled n/2 to represent the subarray sizes for the two subproblems of size n/2.

Each of the subproblems of size n/2 recursively sorts two subarrays of size ((n/2)/2, or n/4. Because there are two subproblems of size n/2, there are four subproblems of size n/4. Each of these four subproblems merges a total of n/4 elements, and so the merging time for each of the four subproblems is cn/4. Summed up over the four subproblems, we see that the total merging time for all subproblems of size n/4 is ***4·cn/4=cn:***

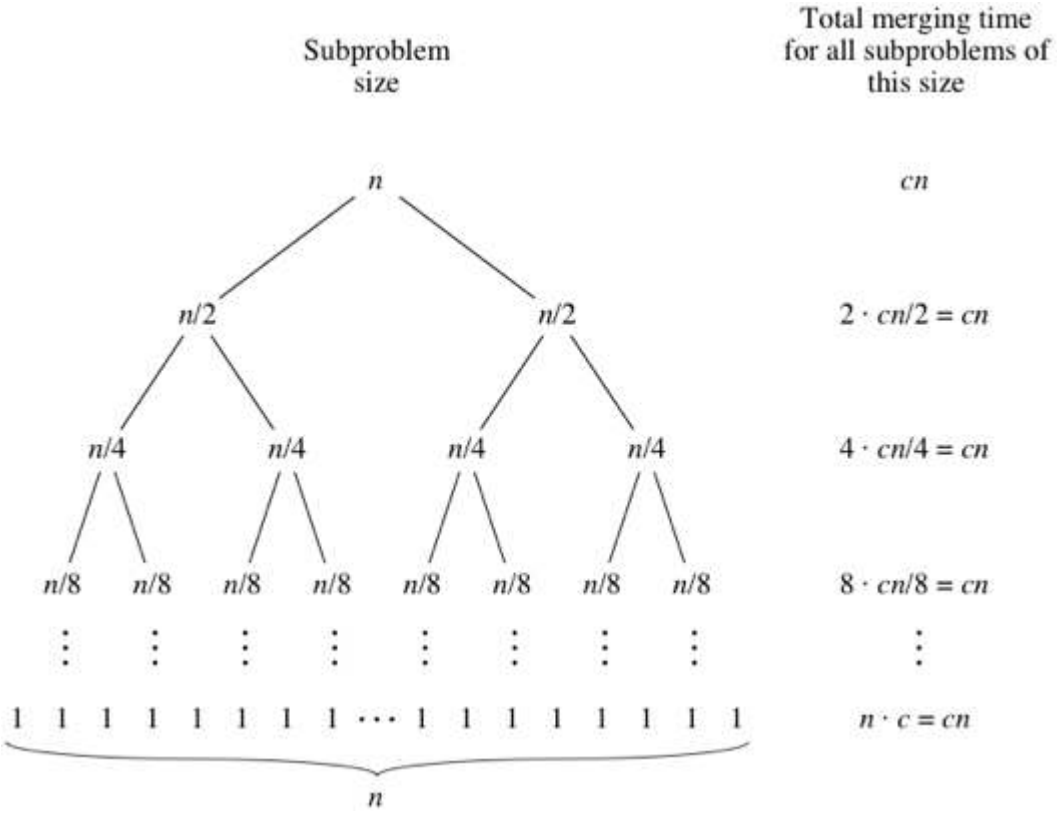


What do you think would happen for the subproblems of size n/8? There will be eight of them, and the merging time for each will be cn/8, for a total

merging time of **8·cn/8=cn**:



As the subproblems get smaller, the number of subproblems doubles at each "level" of the recursion, but the merging time halves. The doubling and halving cancel each other out, and so the total merging time is cn at each level of recursion. Eventually, we get down to subproblems of size 1: the base case. We have to spend Θ(1) time to sort subarrays of size 1, because we have to test whether p < r, and this test takes time. How many subarrays of size 1 are there? Since we started with n elements, there must be n of them. Since each base case takes Θ(1) time, let's say that altogether, the base cases take cn time:



Now we know how long merging takes for each subproblem size. The total

time for mergeSort is the sum of the merging times for all the levels. If there

are ll levels in the tree, then the total merging time is $l \cdot cn$. So what is l ? We start with subproblems of size n and repeatedly halve until we get down to subproblems of size 1. We saw this characteristic when we analyzed binary search, and the answer is $l = lgn+1$. For example, if n=8, then $lgn+1=4$, and sure enough, the tree has four levels: n = 8, 4, 2, 1. The total time for mergeSort, then, is $cn(lgn+1)$. When we use big-Θ notation to describe this running time, we can discard the low-order term (+1) and the constant coefficient (c), giving us a running time of **Θ(nlgn)**, as promised.

One other thing about merge sort is worth noting. During merging, it makes a copy of the entire array being sorted, with one half in lowHalf and the other half in highHalf. Because it copies more than a constant number of elements at some time, we say that merge sort does not work in place. By contrast, both selection sort and insertion sort do work in place, since they never make a copy of more than a constant number of array elements at any one time. Computer scientists like to consider whether an algorithm works in place, because there are some systems where space is at a premium, and thus in-place algorithms are preferred.