Recurrence Part II

This chapter continues the discussion on analyzing the complexity of recursive algorithms.

Determining Recursion Levels

Now let's take an example of how we get to the base case, starting with an array of size 8. We'll first divide it into half, creating two arrays of size 4. Next, we'll split each of the two arrays of size 4 into two arrays of size 2. Now we have four arrays of size 2 but we haven't hit the recursion's base case, so we'll split each of the four arrays of size 2 into a single array of size 1. When we finally hit the recursion's base case, we will have eight arrays of a single element.

Note, we need to find the number of levels or recursions it took to break the problem of size 8 into a problem of size 1. At each recursion level, we divide the problem by 2. So we may ask, *how many times do we need to divide the input size by 2 to get to a problem of size 1?*. For 8 we know we need to divide thrice by 2 to get to a problem size of 1. In other words, we can ask, *2 raised to what power would yield 8?*

$$2^x = 8$$

The above equation is in exponential form and can be expressed in *logarithmic* form as follows:

$$log_2(8) = x$$

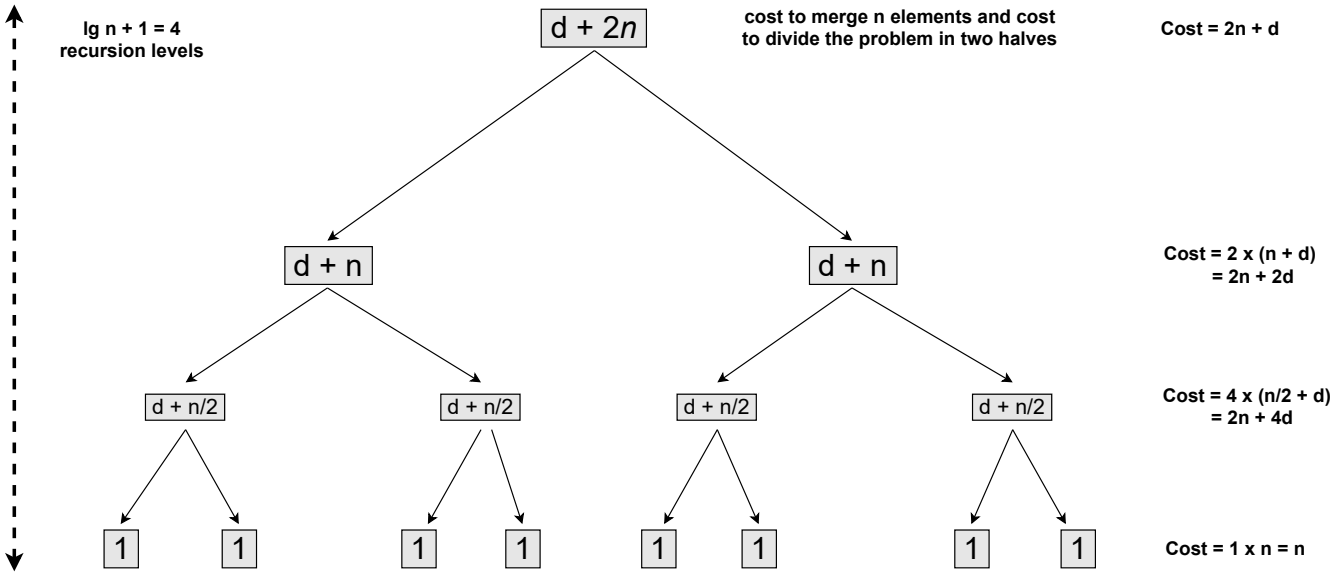
$$log_2(8) = 3$$

Note in our case, the number of recursion levels is $\mathbf{log_2 n + 1}$. We need to add 1 for the root level.

Knowing the number of recursion levels a given input results in, we can

determine the running time of the algorithm by multiplying the number of recursion levels with the cost incurred at each level.

The illustration below shows the recursion levels and the total cost at each level, where ***n*** is the input size and ***d*** is the cost to divide the problem into two subproblems.



Analyzing Cost of Merge Sort

At the base case, the cost to merge a single element array is constant. But we have 8 such base cases, one for each element of the array. If we use n as the initial input size we can see that the cost of the base case is n x O(1) = n. Now, as we return from the base case recursion, we can calculate the cost at the level immediately above the base case. Here, we'll be merging 2 arrays of 1 element in each of the four calls. Expressed in terms of the initial input size, we merge n/4 elements in each call for a total of 4 calls. The cost is thus 4 x n/4 = n. Similarly, at the root level above, we'll be merging n/2 elements in 2 calls for a total cost of 2 x n/2 = n. Note, the cost at each level is O(n) for merging the results from the below levels. If we know how many levels exist, we can simply multiply the cost with the levels to get the total cost for running Merge Sort.

Tying it back

So far we have determined that the runtime of Merge Sort is the solution to the following recurrence equation

> $$T(n) = Cost\ to\ divide\ into\ 2\ subproblems + 2 * T(\frac{n}{2}) + Cost\ to\ merge\ 2\ subproblems$$

e.g. 2 subproblems

$$T(n) = d + 2T(\frac{n}{2}) + 2n$$

Additionally, we have determined the cost at each recursion level to be **d + 2n**. Since the number of recursion levels is **log n + 1**, the solution to the recurrence equation is thus:

$$T(n) = (logn + 1) * (d + 2n)$$

$$T(n) = dlogn + d + 2nlogn + 2n$$

$$T(n) = 2nlogn + 2n + dlogn + d$$

Dropping the lower order terms, we are left with

$$T(n) = O(nlogn)$$

Note that we have made some simplifications.

- For one, the cost to divide the problem into two subproblems is not always *d*. For instance, at the second level it is *2d* and at the third level it is *4d* but we just used *d* in our analysis. The divide step would occur at the second last recursion level for a total of n/2 times. In that case, the total cost will be d * $^{n}/_{2}$ - also expressed as *dn* - which is still Θ(n) and doesn't change the big O for the algorithm.
- Our algorithm for merging the two sorted arrays is simplified for explanation-purposes, and can be optimized, but it still won't change the big O for algorithm.

Space Complexity

In the sample code provided, we used an extra scratch array equal to the size of the input. As pointed out earlier, Merge Sort can be implemented in-place. However, it is error-prone and much harder to implement. The space complexity is thus O(n) for our given implementation.