To find the recurrence relation, we first figure out how the base cases apply into the runtime. This is somewhat simple compared to the actual running of the program itself. We begin by noting that, under the algorithm present, that if the array has zero or one elements, the program immediately ends, meaning that the T(n) has a runtime of one. With the initial base case dealt with, we then note the algorithm's operation itself. As the array we take in is being split into three different thirds, this is merely a runtime of 1 as well. When we move onto the actual adding up of all of the split up arrays recursively, we note that it will take 3T(n/3) times as we are recursively adding up everything. However, as we are not merging at the end, this does not require an addition n at the end as well. To put this step-wise, we get the following.

1. If the array has 0 or 1 elements, stop and return a zero or the only element. T(1) = 1
2. Split the array into three approximately equal-sized thirds. 1
3. Add all the thirds together recursively. 3T(n/3)

This, in turn, means that we get the following recurrence relation prior to the solving by substitution.

$$T(n) = \begin{cases} 1 & if\ n\ \leq 1 \\ 3T\left(\frac{n}{3}\right) & if\ n > 1 \end{cases}$$

As we really don't care about the base case when solving by substitution, we calculate via the following:

$$\begin{aligned} T(n) &= 3T\left(\frac{n}{3}\right) \\ &= 9T\left(\frac{n}{9}\right) \\ &= 27T(\frac{n}{27}) \\ &\dots \\ &= 3^iT(\frac{n}{3^i}) \end{aligned}$$

If we then denote that i = lg n, we then note that this is equivalent to:

$$n\ lg_3n \in \theta(n \log_3 n)$$

Therefore, the big theta complexity is $n \log_3 n$.