# A closer look at the complexity analysis of finding the k'th smallest element in two sorted arrays

devblogs.microsoft.com/oldnewthing/20171023-00

October 23, 2017

Raymond Chen

Given two sorted arrays of the same length, with unique elements, find the *k*th smallest element in the combined collection. One solution involves the double binary search. I'll let you go read the article to see how it works. I'm here to dissect the complexity analysis.

> Since every time we remove ¼ elements, the complexity is $O(2*\log(N))$, *i.e.*, $O(\log(N))$.

Okay, that was pretty glib. How did we get there?

If an algorithm reduces the problem to a smaller problem that is half the size, then the number of reduction steps needed to reduce the problem to size 1 is $\lceil \lg N \rceil$. More generally, if you have an algorithm where the ratio of the size of the next iteration to the size of the current iteration is $r$, then the number of reduction steps is $\lceil \log_{1/r} N \rceil$.

In the above example, the ratio is $r = ¾$, so the number of reduction steps is $\lceil \log_{4/3} N \rceil$. We can change the logarithm to base 2 to arrive at $\lceil (\lg N) \div (\lg (4/3)) \rceil \approx \lceil 2.4 \times \lg N \rceil$.

This is not the same as $2 \times \lg N$ given in the original article. My guess is that the original article calculated the conversion factor incorrectly as $\lg 4 = 2$.

But since this is all for order of magnitude calculations, an error in the constant factor is forgiven because order of magnitude ignores constant factors.

But wait, does the formula even apply in the first place?

The formula applies if the reduced problem is the same as the original problem, but with a smaller size. In the case under consideration, the starting point was two equal-sized arrays, but when we're done, we have two unequal-sized arrays: One of the arrays shrunk in half, but the other stayed the same size.

We used the wrong formula!

Consider the second iteration, where we have one small array and one large array. And suppose the second iteration of the algorithm decides to throw away half of the smaller array. We did not throw away a quarter of the elements. We threw away half of the smaller array, which is one third of the total number of elements, which means that we threw away only a sixth!

It gets worse at the third iteration: If we are unlucky and the algorithm decides to throw away half of the tiny array, we discarded only one tenth of the elements.

So in the worst case, we get into a case of diminishing returns, where were throw away less and less from the small array and never make a dent in the big array. Does this algorithm even guarantee termination?

In mathematics, sometimes the way to solve a problem is to convert it to a harder problem, and then solve the harder problem. In this case, the harder problem is "Given two sorted arrays of *possibly-unequal length*, with unique elements, find the *k*th smallest element in the combined collection."

Let $f(n, m)$ represent the number of reduction steps needed to solve the problem, where $n$ and $m$ are the lengths of the two arrays. If you decide to throw away half of the first array, then the number of remaining steps is $f(\frac{1}{2}n, m)$. Similarly, if you decide to throw away half of the second array, then the number of remaining steps is $f(n, \frac{1}{2}m)$. You now have the recursion $f(n, m) = 1 + \max(f(\frac{1}{2}n, m), f(n, \frac{1}{2}m))$.

The solution to this recursion is $f(n, m) = \lceil \lg n \rceil + \lceil \lg m \rceil$.

You can think of the problem this way: You have two arrays, and based on the algorithm, you cut one in half or you cut the other in half, until both arrays are cut down to just one element. It takes $\lceil \lg n \rceil$ cuts to reduce the first array and $\lceil \lg m \rceil$ cuts to reduce the second array. The algorithm tells you which piece to cut next, but regardless of what order the algorithm gives, the total number of cuts is the same.

In the original problem, $n = m = \frac{1}{2}N$. Therefore, the number of reduction steps is $\lceil \lg \frac{1}{2}N \rceil + \lceil \lg \frac{1}{2}N \rceil = 2\lceil \lg \frac{1}{2}N \rceil = 2\lceil \lg N \rceil - 2$, which is $O(\lg N)$.

So the answer given in the article was off by two. This doesn't make any difference when calculating order of magnitude, but it was interesting that the incorrect calculation was so close to the correct one.

Raymond Chen

**Follow**