Finding kth smallest number from n sorted arrays

So, you have n sorted arrays (not necessarily of equal length), and you are to return the kth smallest element in the combined array (i.e the combined array formed by merging all the n sorted arrays)

I have been trying it and its other variants for quite a while now, and till now I only feel comfortable in the case where there are two arrays of equal length, both sorted and one has to return the median of these two. This has logarithmic time complexity.

After this I tried to generalize it to finding kth smallest among two sorted arrays. Here is the question on SO. Even here the solution given is not obvious to me. But even if I somehow manage to convince myself of this solution, I am still curious as to how to solve the absolute general case (which is my question)

Can somebody explain me a step by step solution (which again in my opinion should take logarithmic time i.e O($\log(n_1) + \log(n_2) \ldots + \log(n_N)$ where $n_1, n_2 \ldots n_N$ are the lengths of the n arrays) which starts from the more specific cases and moves on to the more general one?

I know similar questions for more specific cases are there all over the internet, but I haven't found a convincing and clear answer.

Here is a link to a question (and its answer) on SO which deals with 5 sorted arrays and finding the median of the combined array. The answer just gets too complicated for me to able to generalize it.

Even clean approaches for the more specific cases (as I mentioned during the post) are welcome.

PS: Do you think this can be further generalized to the case of unsorted arrays?

PPS: It's not a homework problem, I am just preparing for interviews.

arrays algorithm data-structures merge

edited Jan 6 '12 at 13:48

Saeed Amiri
17k 4 20 61

asked Jan 6 '12 at 4:25

Sushant
337 3 16

This question is more suited for StackProgramming, I think. I can't answer your question anyhow :) – Daryl Teo Jan 6 '12 at 5:19

1 What do you mean by logarithmic time? We have two parameters, n and k. I don't think you can get faster than O(n) because you will have to look at each array at least once. – David Grayson Jan 6 '12 at 5:19

logarithmic means something like O(lg(n1) + lg(n2) + lg(n3)...) where n1, n2, n3.. are the lengths of arrays n1, n2, n3...n – Sushant Jan 6 '12 at 5:29

7 Answers

This doesn't generalize the links, but does solve the problem:

1. Go through all the arrays and if any have length > k, truncate to length k (this is silly, but we'll mess with k later, so do it anyway)
2. Identify the largest remaining array A. If more than one, pick one.
3. Pick the middle element M of the largest array A.
4. Use a binary search on the remaining arrays to find the same element (or the largest element <= M).
5. Based on the indexes of the various elements, calculate the total number of elements <= M and > M. This should give you two numbers: L, the number <= M and G, the number > M
6. If k < L, truncate all the arrays at the split points you've found and iterate on the smaller arrays (use the bottom halves).
7. If k > L, truncate all the arrays at the split points you've found and iterate on the smaller arrays (use the top halves, and search for element (k-L).

When you get to the point where you only have one element per array (or 0), make a new array of size n with those data, sort, and pick the kth element.

Because you're always guaranteed to remove at least half of one array, in *N* iterations, you'll get rid of half the elements. That means there are *N log k* iterations. Each iteration is of order *N log k* (due to the binary searches), so the whole thing is *N^2 (log k)^2* That's all, of course, worst case, based on the assumption that you only get rid of half of the largest array, not of the other arrays. In practice, I imagine the typical performance would be quite a bit better than the worst case.

answered Jan 10 '12 at 6:50

Michael
719 3 4

Neat. Thanks... – Sushant Jan 11 '12 at 3:35

2 Don't you think it can be solved in N^2LogN by simple min heap. Take a heap of size N and put the minimum elements form the N arrays then pop one and check for the array. Insert the next element from the same array. Keep on doing this quit once you got you Kth element from the heap. – Trying Jan 19 '14 at 4:40

geeksforgeeks.org/... This solution shows that it can be done in O(N + kLogN) time. – dilip Nov 16 '14 at 4:31

It can not be done in less than `O(n)` time. *Proof Sketch* If it did, it would have to completely not look at at least one array. Obviously, one array can arbitrarily change the value of the `kth` element.

I have a relatively simple `O(n*log(n)*log(m))` where `m` is the length of the longest array. I'm sure it is possible to be slightly faster, but not a lot faster.

Consider the simple case where you have `n` arrays each of length 1. Obviously, this is isomorphic to finding the `k` th element in an unsorted list of length `n` . It is possible to find this in `O(n)` , see Median of Medians algorithm, originally by Blum, Floyd, Pratt, Rivest and Tarjan, and no (asymptotically) faster algorithms are possible.

Now the problem is how to expand this to longer sorted arrays. Here is the algorithm: Find the median of each array. Sort the list of tuples `(median,length of array/2)` and sort it by median. Walk through keeping a sum of the lengths, until you reach a sum greater than k. You now have a pair of medians, such that you know the kth element is between them. Now for each median, we know if the kth is greater or less than it, so we can throw away half of each array. Repeat. Once the arrays are all one element long (or less), we use the selection algorithm.

Implementing this will reveal additional complexities and edge conditions, but nothing that increases the asymptotic complexity. Each step

1. Finds the medians or the arrays, `O(1)` each, so `O(n)` total
2. Sorts the medians `O(n log n)`
3. Walks through the sorted list `O(n)`
4. Slices the arrays `O(1)` each so, `O(n)` total

that is `O(n) + O(n log n) + O(n) + O(n) = O(n log n)` . And, we must perform this untill the longest array is length 1, which will take `log m` steps for a total of `O(n*log(n)*log(m))`

You ask if this can be generalized to the case of unsorted arrays. Sadly, the answer is no. Consider the case where we only have one array, then the best algorithm will have to compare at least once with each element for a total of `O(m)`. If there were a faster solution for n unsorted arrays, then we could implement selection by splitting our single array into n parts. Since we just proved selection is `O(m)`, we are stuck.

answered Jan 6 '12 at 6:34

Philip JF

18.5k 3 46 62

This is just a specific case where k = n/2 i.e finding the median is equivalent to finding the n/2th smallest overall. This particular problem can be solved by finding the medians of each array (which is O(1) because the arrays are sorted. Then, finding the min and the max of these n medians O(n) time. Now the combined median will lie between the min and max median, so we can get rid of other elements. This is essentially O(log(maxM)) but I am not sure. In your case sorting the medians is bringing up the complexity a little bit when all we need would be a min and max. +1 for the effort though – Sushant Jan 6 '12 at 6:51

Yah, going from finding the median to finding the kth is not hard. The problem with just picking the min and the max is you dont have bounds on how many values you are throwing away, unless I'm missing something. The sorting step lets you throw away half the values. Still, it would be nice to get down to `O(n*log m)` – Philip JF Jan 6 '12 at 8:37

Note that my solution has the same asymptotic behavior as the one you linked to. Since in that case `n = 5` it was treated as a contant for the big-Oh calculation. – Philip JF Jan 6 '12 at 8:42

With the knowledge that the median is bw min and max, you can certainly throw the lower half of the array corresponding to the min median, and upper half for the max median. – Sushant Jan 7 '12 at 3:01

And how can this scheme be easily extended to finding kth? – Sushant Jan 7 '12 at 3:25

You could look at my recent answer on the related question here. The same idea can be generalized to multiple arrays instead of 2. In each iteration you could reject the second half of the array with the largest middle element if k is less than sum of mid indexes of all arrays. Alternately, you could reject the first half of the array with the smallest middle element if k is greater than sum of mid indexes of all arrays, adjust k. Keep doing this until you have all but one array reduced to 0 in length. The answer is kth element of the last array which wasn't stripped to 0 elements.

Run-time analysis:

You get rid of half of one array in each iteration. But to determine which array is going to be reduced, you spend time linear to the number of arrays. Assume each array is of the same length, the run time is going to be c*c*log(n), where c is the number of arrays and n is the length

of each array.

answered Jan 27 '12 at 17:16

lambdapilgrim
449 4 5

If the k is not that huge, we can maintain a priority min queue. then loop for every head of the sorted array to get the smallest element and en-queue. when the size of the queue is k. we get the first k smallest .

maybe we can regard the n sorted array as buckets then try the bucket sort method.

answered Jan 6 '12 at 5:23

hxc
6 4

> What is the complexity? In addition to using O(k) space, you will do atleast K enqueues, which is O(k log(k)). So, yep if K is huge we have a problem. I agree it might be a good solution when you need all the k smallest numbers, but in this case I just need the kth smallest. – Sushant Jan 6 '12 at 5:36

This could be considered the second half of a merge sort. We could simply merge all the sorted lists into a single list...but only keep k elements in the combined lists from merge to merge. This has the advantage of only using O(k) space, but something slightly better than merge sort's O(n log n) complexity. That is, it should in practice operate slightly faster than a merge sort. Choosing the kth smallest from the final combined list is O(1). This is kind of complexity is not so bad.

answered Jan 6 '12 at 6:25

Andy
1

> Yes this is one of the trivial solutions to the problem, unfortunately it is not optimal enough. – Sushant Jan 6 '12 at 6:40

There exist an generalization that solves the problem in O(N log k) time, see the question here.

answered Feb 9 '14 at 3:08

Chao Xu

883 7 22

Please find the below C# code to Find the k-th Smallest Element in the Union of Two Sorted Arrays. Time Complexity : O(logk)

```
public int findKthElement(int k, int[] array1, int start1, int end1, int[] array2, int
start2, int end2)
    {
        // if (k>m+n) exception
        if (k == 0)
        {
            return Math.Min(array1[start1], array2[start2]);
        }
        if (start1 == end1)
        {
            return array2[k];
        }
        if (start2 == end2)
        {
            return array1[k];
        }
        int mid = k / 2;
        int sub1 = Math.Min(mid, end1 - start1);
        int sub2 = Math.Min(mid, end2 - start2);
        if (array1[start1 + sub1] < array2[start2 + sub2])
        {
            return findKthElement(k - mid, array1, start1 + sub1, end1, array2, start2,
end2);
        }
        else
        {
            return findKthElement(k - mid, array1, start1, end1, array2, start2 + sub2,
end2);
        }
    }
```

answered Aug 11 '14 at 22:13

Piyush Patel

37 9
