given 2 sorted arrays of integers, find the nth largest number in sublinear time [duplicate]

Possible Duplicate:
How to find the kth smallest element in the union of two sorted arrays?

This is a question one of my friends told me he was asked while interviewing, I've been thinking about a solution.

Sublinear time implies logarithmic to me, so perhaps some kind of divide and conquer method. For simplicity, let's say both arrays are the same size and that all elements are unique

algorithm

edited Jan 14 '11 at 6:33
Aryabhatta

asked Jan 14 '11 at 0:19

Dan Q
586 2 7 25

marked as duplicate by Moron, Matthieu M., marcog, larsmans, Alejandro Jan 14 '11 at 17:25

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

I don't think this is possible but I'll be happy to be proved wrong which is why this is a comment rather than an answer :-) The only way way to make the extract itself O(1) is to merge the lists which is O(n). Any other solution is missing the vital bit of information as to how the integers are allocated among the two lists hence you have to examine all the values in them (above the one you're looking for) - this automatically makes it O(n) linear. But I'll give you +1 for an interesting question nonetheless. – paxdiablo Jan 14 '11 at 0:29

@paxdiablo- Can you check my answer and see if I'm missing something? I seem to have gotten this in O(lg^2 n). – templatetypedef Jan 14 '11 at 0:41

add a comment

5 Answers

I think this is two concurrent binary searches on the subarrays `A[0..n-1]` and `B[0..n-1]`, which is O(log n).

- Given sorted arrays, you know that the *nth* largest will appear somewhere before or at `A[n-1]` if it is in array `A`, or `B[n-1]` if it is in array `B`
- Consider item at index `a` in `A` and item at index `b` in `B`.
- Perform binary search as follows (pretty rough pseudocode, not taking in account 'one-off' problems):
  - If `a + b > n`, then reduce the search set
    - if `A[a] > B[b]` then `b = b / 2`, else `a = a / 2`
  - If `a + b < n`, then increase the search set
    - if `A[a] > B[b]` then `b = 3/2 * b`, else `a = 3/2 * a` (halfway between `a` and previous `a`)
  - If `a + b = n` then the *nth* largest is `max(A[a], B[b])`

I believe worst case O(ln n), but in any case definitely sublinear.

edited Jan 14 '11 at 1:47

answered Jan 14 '11 at 1:29

Kirk Broadhurst

13.6k 4 35 69

I really like this idea! Do you have a correctness proof? I'll try working one out if you don't. – templatetypedef Jan 14 '11 at 1:55

1 Methinks that you don't want to scale up by 3/2. I think what you probably want to do is just a standard binary search over both ranges starting in the middle of each and maintain an a_low, a_high, and a_mid value (plus one of each for b). I'll get back to you if this works. – templatetypedef Jan 14 '11 at 2:17

@templatetypedef Yes, that is what I originally intended to do but I didn't want to have to include the low/high stuff in my algorithm - feeling kind of lazy! Definnitely preferable though. – Kirk Broadhurst Jan 14 '11 at 2:44

What are your starting indexes (a and b) in the arrays? – r0u1i Jan 14 '11 at 14:48

@r0u1i Starting indexes are `a = b = n - 1` , because the *nth* largest must appear within the first `n - 1` elements. – Kirk Broadhurst Jan 17 '11 at 0:25

show **2** more comments

I believe that you can solve this problem using a variant on binary search. The intuition behind this algorithm is as follows. Let the two arrays be A and B and let's assume for the sake of simplicity that they're the same size (this isn't necessary, as you'll see). For each array, we can construct parallel arrays Ac and Bc such that for each index i, Ac[i] is the number of elements in the two arrays that are no larger than A[i] and Bc[i] is the number of elements in the two arrays that are no larger than B[i]. If we could construct these arrays efficiently, then we could find the kth smallest element efficiently by doing binary searches on both Ac and Bc to find the value k. The corresponding entry of A or B for that entry is then the kth largest element. The binary search is valid because the two arrays Ac and Bc are sorted, which I think you can convince yourself of pretty easily.

Of course, this solution doesn't work in sublinear time because it takes O(n) time to construct the arrays Ac and Bc. The question then is - is there some way that we can *implicitly* construct these arrays? That is, can we determine the values in these arrays without necessarily constructing each element? I *think* that the answer is yes, using this algorithm. Let's begin by searching array A to see if it has the kth smallest value. We know for a fact that the kth smallest value can't appear in the array in array A after position k (assuming all the elements are distinct). So let's focus just on the the first k elements of array A. We'll do a binary search over these values as follows. Start at position k/2; this is the k/2th smallest element in array A. Now do a binary search in array B to find the largest value in B smaller than this value and look at its position in the array; this is the number of elements in B smaller than the current value. If we add up the position of the elements in A and B, we have the total number of elements in the two arrays smaller than the current element. If this is exactly k, we're done. If this is less than k, then we recurse in the upper half of the first k elements of A, and if this is greater than k we recurse in the lower half of the first elements of k, etc. Eventually, we'll either find that the kth largest element is in array A, in which case we're done. Otherwise, repeat this process on array B.

The runtime for this algorithm is as follows. The search of array A does a binary search over k elements, which takes O(lg k) iterations. Each iteration costs O(lg n), since we have to do a binary search in B. This means that the total time for this search is O(lg k lg n). The time to do this in array B is the same, so the net runtime for the algorithm is O(lg k lg n) = $O(\lg^2 n)$ = o(n), which is sublinear.

answered Jan 14 '11 at 0:34

templatetypedef

142k 31 340 572

1 That sounded interesting enough I had to try it out: gist.github.com/779003 – Nemo157 Jan 14 '11 at 1:46

Just a small addition, you don't have to search through the entire B array either, once the binary search takes you strictly above or under index k/2, you already know the answer to whether the sum is greater than k. – biziclop Jan 14 '11 at 10:17

add a comment

```
int[] a = new int[] { 11, 9, 7, 5, 3 };
int[] b = new int[] { 12, 10, 8, 6, 4 };
int n = 7;
int result = 0;
if (n > (a.Length + b.Length))
    throw new Exception("n is greater than a.Length + b.Length");
else if (n < (a.Length + b.Length) / 2)
{
    int ai = 0;
    int bi = 0;
    for (int i = n; i > 0; i--)
    {
        // find the highest from a or b
        if (ai < a.Length)
        {
            if (bi < b.Length)
            {
                if (a[ai] > b[bi])
                {
                    result = a[ai];
                    ai++;
                }
                else
                {
                    result = b[bi];
                    bi++;
                }
            }
            else
            {
                result = a[ai];
                ai++;
            }
        }
        else
        {
```

edited Jan 14 '11 at 1:06

answered Jan 14 '11 at 0:37

Tim Carter
514 2 9

add a comment

This is quite similar answer to Kirk's.

Let `Find( nth, A, B )` be function that returns nth number, and |A| + |B| >= n. This is simple pseudo code without checking if one of array is small, less than 3 elements. In case of small array one or 2 binary searches in larger array is enough to find needed element.

```
Find( nth, A, B )
  If A.last() <= B.first():
    return B[nth - A.size()]
  If B.last() <= A.first():
    return A[nth - B.size()]
  Let a and b indexes of middle elements of A and B
  Assume that A[a] <= B[b] (if not swap arrays)
  if nth <= a + b:
    return Find( nth, A, B.first_half(b) )
  return Find( nth - a, A.second_half(a), B )
```

It is `log(|A|) + log(|B|)` , and because input arrays can be made to have n elements each it is `log(n)` complexity.

answered Jan 14 '11 at 11:22

Ante
2,363 4 12 25

add a comment

Sublinear of what though? You can't have an algorithm that doesn't check at least n elements, even verifying a solution would require checking that many. But the size of the problem here should surely mean the size of the arrays, so an algorithm that only checks n elements is sublinear.

So I think there's no trick here, start with the list with the smaller starting element and advance until you either:

1. Reach the nth element, and you're done.
2. Find the next element is bigger than the next element in the other list, at which point you switch to the other list.
3. Run out of elements and switch.

answered Jan 14 '11 at 0:35

biziclop
22.1k 4 35 54

3 -1 Binary search doesn't read every element. – marcog Jan 14 '11 at 0:44

Of course not, but to select the nth element from a list of any size, you have to read at least n elements. – biziclop Jan 14 '11 at 0:47

4 @biziclop- If you have a sorted array, you can pick the nth largest in O(1) by just indexing at position n. – templatetypedef Jan 14 '11 at 0:49

Technically for a *list* he's correct that reading the nth element is O(n), however this is using sorted *arrays* which are O(1). – Nemo157 Jan 14 '11 at 3:10

@templatetypedef If both arrays can be indexed at position n in O(1), you're still back at square one with two arrays of length n. – Apalala Jan 14 '11 at 5:35

show **1** more comment

Not the answer you're looking for? Browse other questions tagged algorithm or ask your own question.