

Question 1. If $n = 1$ then return the smaller of $A[0]$ and $B[0]$. Otherwise compare $A[(n/2) - 1]$ to $B[(n/2) - 1]$:

- If $A[(n/2) - 1] > B[(n/2) - 1]$ then recursively solve (and return the answer from) the problem that consists of the left half of A and right half of B (each half is of size $n/2$, and we seek the $(n/2)$ th smallest element among them).
- If $A[(n/2) - 1] < B[(n/2) - 1]$ then recursively solve (and return the answer from) the problem that consists of the right half of A and left half of B (each half is of size $n/2$, and we seek the $(n/2)$ th smallest element among them).

Why it works. After a comparison, the half of an array that we “throw away” (in the sense that it is not part of the recursive call performed next) consists of items that are known to be (i) larger than n other items (hence “too large” to be the answer), or (ii) smaller than $n + 1$ other items (hence “too small” to be the answer); in either case the items “thrown away” are no longer candidates to be the answer we seek (so it is correct to throw them away). The reason we seek the $(n/2)$ th item in the recursive call (this is implicit to the recursive call) is that, before recursing, we threw away $n/2$ items that are smaller than the answer we seek, so the answer we seek is now $(n/2)$ th smallest (in the subproblem we’re recursing on) rather than n th smallest.

Why it runs in $O(\log n)$ time. For the same reason that binary search runs in $O(\log n)$ time: After a comparison, the problem size has been reduced by a factor of two, i.e., we have $T(n) = T(n/2) + c$ and $T(1) = c'$ where c and c' are constants (that the solution to this is $O(\log n)$ can be seen either by solving it, or by observing that the recursion tree is unary, has logarithmic height, and has a single comparison performed at each of its nodes).

Question 2.

1. The space is $O(kn \log k)$ because each of the kn items appears $\log k$ times (once at each level of T).
2. It also takes $O(kn \log k)$ time to build because, if u and w are children of v then the list at v (call it $L(v)$) can be obtained in time proportional to its size by merging $L(u)$ and $L(w)$. So the time complexity is same as the space complexity.
3. During the above-mentioned merge we also compute the “cross-rank” information between $L(v)$ and each of $L(u)$ and $L(w)$: Every item of $L(v)$ stores links for determining in constant time its position in $L(u)$ and $L(w)$. To locate x in the k lists at the leaves, we first do a logarithmic-time binary search for x in the list at the root, after which we go down one level at a time: we locate x in constant time in each of the children of the root (constant time for each child because we just follow a cross-rank link to that child’s list), then in each of the grandchildren of the root (again in constant time for each grandchild), etc (until we reach the leaves). The key idea is that once we know the location of x in an $L(v)$, we can in constant time get the location of x in $L(u)$ and $L(w)$ for the children u, w of v . Therefore, once we have done the $O(\log kn)$

time binary search at the root, we can complete the job in an additional $O(k)$ time because T has $O(k)$ nodes. The overall time is $O(\log kn + k)$, which is $O(\log n + k)$.

4. To improve the above to $O(kn)$ space and $O(kn)$ construction time for the data structure, we modify the way $L(v)$ is obtained from the lists of its two children: If v has children u and w then we define $L(v)$ as the merge of $L'(u)$ and $L'(w)$, where $L'(u)$ is obtained from $L(u)$ by choosing every other element of $L(u)$ ($L'(w)$ is similarly obtained from $L(w)$). Therefore for any v , $L'(v)$ has half as many elements as $L(v)$. The total size of the list at a parent of leaves is therefore $kn/2$, at a grandparent of leaves it is $kn/4$, etc. The overall space (and hence the time to build the structure) is therefore $(nk) \sum_i 2^{-i} = O(nk)$.

Of course in this “slimmed down” version of T by following a cross-rank link we locate x in an $L'(v)$ rather than in an $L(v)$, but obtaining the position of x in $L(v)$ when we already know its position in $L'(v)$ takes constant time (1 extra comparison).