ADA 2022 Tutorial 2

February 3 2022

1 Finding Majority

You are given an array of n elements, not necessarily distinct. An element is called *majority* if it appears strictly more than $n/2$ times. Your task is to find a majority if it exists. Importantly, the only operation you are allowed to perform to compare two elements is to test if the two elements are identical or different, i.e. there is no notion of an element being less than or greater than another element (and hence *you cannot sort*!), For instance, say the elements are DNA samples, and you can only test if two samples match, there is no notion of ordering, $<$ or $>$ on the samples.

a. Design a *divide-and-conquer* algorithm to find the majority element or report there exists none. Your algorithm should perform $\mathcal{O}(n \log n)$ tests.

Solution. The following is the recursive algorithm at a high level.

(a) If A has 3 or less elements, just return majority by counting. Else do the following.
(b) Divide the given array A at the middle - call the left half L and the right half R.
(c) Recursively find the majority in L - call it ℓ, null if no majority and majority in R - call it r - , null otherwise.
(d) If null returned in both recursive calls, return 'no majority'
(e) Else do the following. Count the number of times ℓ appears in R. If the sum of count from the recursive call on L and this count exceeds $n/2$, return *ell* and the total count. Else repeat the analogous thing for r.
(f) If none of the counts above exceeds $n/2$ return null.

The combine step clearly runs in $\Theta(n)$ time since all we are doing is going over each half linearly and comparing with a fixed element. Hence, the recursion is $T(n) = 2T(n/2) + \mathcal{O}(n)$ which solves to $\mathcal{O}(n \log n)$

Correctness.

Claim 1 *The above algorithm returns the majority in array A and its count correctly if and only if such such an element exists.*

Proof: We prove this by induction on the size of the array. The base cases are easy to check. Now suppose claim holds for all arrays of size $1, 2, \cdots n-1$. Now consider A which is of size n. By induction hypothesis, we know that the algorithm returns correct answer for the subarrays L and R. Now we prove the two directions.

- Suppose there exists a majority element x in A. Then, by Pigeon Hole Principle, either L or R must contains strictly more than $n/4 + 1$ copies of x. Hence, the algorithm correctly enters step (d). Now it is easy to see that the algorithm will return the correct answer for A.
- Suppose there is no majority in A. Now there could be two cases. If null is returned for both calls in step (c), then we are done by induction hypothesis. Suppose that does not happen and algorithm enters step (d). (Note: This can happen. There might be a majority element in either L or R even if there is no majority in A). In this case, both the counts has to return less than $n/2 + 1$. Again, note that we are implicitly using induction hypothesis on arrays L and R of size $n/2$ each when we are assuming that they return the majority count correctly.

□

b. * Do the above in $\mathcal{O}(n)$ tests. **Solution**

There are two ways that I know of for doing this. I am showing you one of them which I feel uses the spirit of divide and conquer, although its not the usual divide and conquer thing. I am assuming n is a power of 2 for simplicity.

(a) If A contains one element return this element as majority. Else do the following

(b) Keep making disjoint pairs by testing adjacent elements. If they are different, remove them from A. Otherwise replace both copies with a single copy of the common element. In case the array length is odd, just copy the last element as it is.

(c) Recurse on the modified array.

(d) Suppose the above recursion returns x. Now make a final pass through the original array A to check if x is indeed the majority. (This is a super important step !!!!)

See figure below for an example run.

Let us introduce the notation A_ℓ to denote the array after ℓ recursive calls. For example, $A_0 = A$.

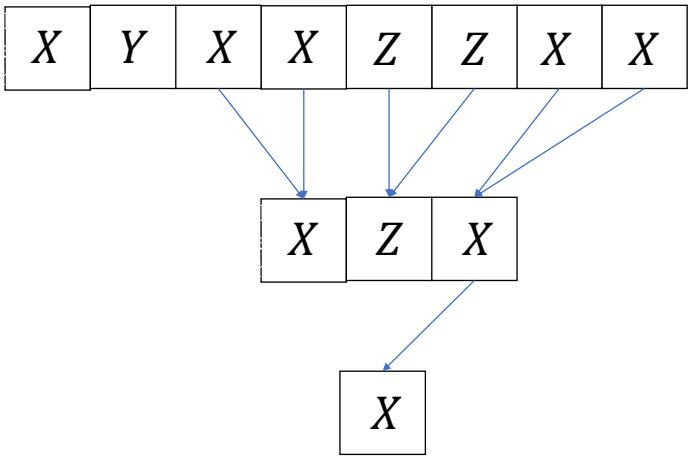
Runtime: Its clear that the runtime for level ℓ is $\mathcal{O}(|A_\ell) = \mathcal{O}(n/2^\ell)$ since all we are doing is a linear scan through the array and comparing adjacent elements. Note that at every step, you are reducing the size of the array by at least half and hence $|A_\ell| \leq n/2^\ell$. So, the total runtime is $\sum_{\ell=0}^{\log n} c \cdots \frac{n}{2^\ell} \leq 2cn = \mathcal{O}(n)$. You need another $O(n)$ for the final pass.

Correctness

Claim 2 *An element x survives in $A_{\log n}$ if x is a majority in A.*

Proof: Suppose x is a majority in A. We show by induction on ℓ that x is a majority in $A_\ell, \forall \ell = 0, 1, 2, \cdots \log n - 1$. The base case is trivially true. Now consider a level $\ell + 1$ and assume by induction hypothesis that x is majority in A_ℓ. Suppose for the sake of contradiction x is not so in $A_{\ell+1}$. Now, let us count the number of occurrences of x in A_ℓ. Well, each occurrence of x can account for two occurrences of x in A_ℓ. On the other hand, each deleted pair from A_ℓ accounts for at most 1 occurrence of x. Thus, x cannot be a majority in A_ℓ leading to a contradiction. (Note the subtle points where we are using that pairs are disjoint - do you see where ?).

□



Note the very subtle point here. The above proof shows that the recursive part of the algorithm returns a majority element correctly *if* such an element exists. However, it might return a wrong element as majority in case there is no majority element!. This is why you need a final pass in order to check whether the returned element is indeed majority or not. As an exercise, find an example where the recursive part can spit out a wrong answer in case there is no majority.

2 Finding n-th smallest element of two sorted arrays

Suppose you are given two *sorted* arrays A and B, each of size n. Design an $\mathcal{O}(\log n)$ algorithm to find the n-th smallest element of the union of A and B

Solution. We apply a classical divide and conquer approach. Here is the intuition. Suppose we compare the middle elements of the arrays A and B (recall they are sorted and hence this takes constant time). Suppose $A[mid] > B[mid]$ (the other direction is symmetric). Then what do we conclude about the location of the nth smallest element ? Well, it cannot lie in the second half of A, since there are already n elements which are smaller than any element from that half - namely, all elements in the first half of A (including the middle) and all elements in the first half of B). Further, it cannot lie in the first half of B since those elements are strictly less than $A[mid]$ and hence lie among the first $n-1$ elements in the whole array. Hence, we continue our hunt for the n-th smallest element in two subarrays - the first half of A and the second half of B. But wait ! Which element should I look for now ? Well we just look for the $n/2$th smallest element in the combined array of these two subarrays. Why ? Because we already have $n/2$ elements that lie among the first $n-1$ elements of the combined array. So it is enough to find the $n/2$th smallest element in the remaining search space.

3 Local minimum in an array

1.* Given an array A, we say that an element $A[i]$ is a local minimum if $A[i] \leq A[i+1]$ and $A[i] \leq A[i-1]$ (we only check the inequality for those elements $A[i+1], A[i-1]$ which exist). In other words, a local minimum is an element which is less than or equal to each of its (at most 2) neighbors. Give an algorithm to find *any* local minimum in an array of n elements by making $\mathcal{O}(\log n)$ comparisons.

Solution. First notice that a local minimum always exists - the global minimum (i.e. the smallest element) is always a local minimum. Naively finding it would need $O(n)$ comparisons, but we can do better since we only need *any* local minimum, not necessarily the global minimum. The idea, as one might guess, is to use binary search.

Consider the middle element $A[n/2]$ (assume n is a power of 2). If $A[n/2]$ is a local minimum, just return it. Else it partitions the array into two halves L, R. But the question is - *in which half can we be sure to find a local minimum ?*

The idea is to decide locally i.e. if the left (resp. right) neighbor of $A[n]$ is smaller than $A[n]$, then we should look in the left (resp. right) subarray. More formally, consider algorithm 1.

Algorithm 1 Algorithm for local min in an array

```
1:  procedure LocalMin(A)                          ▷ Local Min of array of length n
2:      If A has size at most 3, find a local minimum by brute force. Otherwise,
3:      if A[n/2] ≤ A[n/2 − 1] and A[n/2] ≤ A[n/2 + 1] then return n/2
4:      else if A[n/2] > A[n/2 − 1] then           ▷ Look for a local min on the left
5:          L ← A[1] ... A[n/2]
6:          return LocalMin(L)
7:      else                                       ▷ Look for a local min on the right
8:          R ← A[n/2] ... A[n]
9:          return LocalMin(R)
10:     end if
```

It is easy to see that the algorithm 1 makes only $\mathcal{O}(\log n)$ comparisons, similar to binary search, the recurrence is $T(n) = T(n/2) + O(1)$.

Let us see why it will always find a local minimum. Consider the case when $A[n/2]$ is not a local minimum. Then, the algorithm works because the local min x from the recursive call will not be $A[n/2]$. This is because we choose the half to recurse on so that this happens. Hence, x will have the same neighbors in the subarray recursed on, as its neighbors in A. Hence, x will also be a local minimum in A.

Formal proof by induction:

$P(n)$: the algorithm 1 finds a local minimum for any array of size n.

Base case: $P(1), P(2), P(3)$ hold - If A has at most 3 elements, we are done by brute force.

Induction hypothesis: Suppose $P(k)$ holds whenever $k < n$.

Induction Step: If $A[n/2]$ is a local minimum, the algorithm will return it. Otherwise, either $A[n/2] > A[n/2-1]$ or $A[n/2] > A[n/2+1]$. Let us focus on the first case (the other case is symmetric). Then, by applying the induction hypothesis on the length of L, the algorithm returns a local min x in L. x cannot be $A[n/2]$ (because $A[n/2]$ is bigger than its left neighbor

in L). Since x is not the rightmost element of L, the neighbors of x in L are the same as the neighbors of x in A. Hence, x is also a local min in A.

2.** Given an $m \times n$ 2-D array A, we say that an element $A[i][j]$ is a local minimum if it is no more than each of its (at most four) neighbors i.e. $A[i][j] \leq A[i+1][j]$, $A[i][j] \leq A[i][j+1]$, $A[i][j] \leq A[i-1][j]$ and $A[i][j] \leq A[i][j-1]$ (again, we only check the inequality for those elements which exist). Give an algorithm to find any local minimum in A by making $\mathcal{O}(n \log m)$ comparisons.

Solution. The following observation is the key to everything:

Lemma 3 *Consider the array B of length n defined as $B[j] = \min_i A[i][j]$. Then, a local min in B is a local min in A.*

Proof: Suppose $B[j]$ is a local minimum in B. Now by definition of $B[j]$, $A[i][j] \leq A[i+1][j]$ and $A[i][j] \leq A[i-1][j]$. Also, $A[i][j] = B[j] \leq B[j-1] \leq A[i][j-1]$ and $A[i][j] = B[j] \leq B[j+1] \leq A[i][j+1]$, using that $B[j]$ is a local minimum. Hence, $A[i][j]$ is a local minimum in A. □

Now, we can just find a local min in B by using the algorithm 1 for finding a local min in a 1D array. We do not need to find all the entries of B, whenever the algorithm 1 needs an entry $B[j]$ of B, just find the smallest entry of the jth column of A in m comparisons. Since algorithm 1 needs $\mathcal{O}(\log n)$ comparisons of entries of B, the total number of comparisons is $\mathcal{O}(m \log n)$.

This can actually be improved even to $\mathcal{O}(\log m + \log n)$!)