ADA 2024 Tutorial 2

January 2024

1 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.

2 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 **??**.

Algorithm 1 Algorithm for local min in an array

procedure LOCALMIN(A) ▷ Local Min of array of length n
  If A has size at most 3, find a local minimum by brute force. Otherwise,
  if $A[n/2] \leq A[n/2-1]$ and $A[n/2] \leq A[n/2+1]$ **then** return $n/2$
  else if $A[n/2] > A[n/2-1]$ **then** ▷ Look for a local min on the left
    $L \leftarrow A[1] \ldots A[n/2]$
    return LocalMin(L)
  else ▷ Look for a local min on the right
    $R \leftarrow A[n/2] \ldots A[n]$
    return LocalMin(R)
  end if

It is easy to see that the algorithm **??** 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 **??** 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 1 *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 **??** for finding a local min in a 1D array. We do not need to find all the entries of B, whenever the algorithm **??** needs an entry $B[j]$ of B, just find the smallest entry of the jth column of A in m comparisons. Since algorithm **??** 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 $\mathcal{O}(\log m + \log n)$!)

3 Putting Tiles into Checkerboard

Problem: Consider the problem of putting L-shaped tiles (L-shaped consisting of three squares) in an $n \times n$ square-board with exactly one defective square. You can assume that n is a power of 2. Tiles cannot be put in that square. Two L-shaped tiles cannot intersect each other. Describe a divide and conquer based algorithm that computes a proper tiling of the board. Justify the running time of your algorithm.

Solution (sketch): The main idea is the following. Divide and checkerboard into 4 equal checkerboards, each of size $n/2 \times n/2$.

Let $n = 2^k$. Since n is a power of 2, $n/2 = 2^{k-1}$. Let the four checkerboards are A_1, A_2, A_3, A_4.

Note that there is only one checkerboard that has one square defective. Let that checkerboard be A_1. Moreover, the checkerboards are placed in the following order.

$$A_1 || A_2$$
$$A_3 || A_4$$

Observe that the topmost right corner of A_3, the topmmost left corner of A_4 and bottommost left corner of A_2. These three squares together form an L-shaped tile.

The first step is to place an L-shaped tile using these above mentioned three specified squares.

This ensures us that each of the four squares A_1, A_2, A_3, A_4 have exactly one square defective.

Next step is to recursively tile each of these four squares.

This procedure gives us the recurrence

$$T(n) = 4T(n/2) + O(n)$$