COL 351 Aug 21-25, 2023

TUTORIAL SHEET 4

1. We have n nuts and n bolts. The nuts (and the bolts) are of different sizes. Each bolt fits in exactly 1 nut. We would like to match the nuts with the bolts which fits into them. Since the dimensions of the nuts and the bolts are so small, we can not really tell if a nut (or a bolt) is bigger than another nut (or bolt). So the only operation that is allowed is comparing a nut and a bolt – with such a comparison we can distinguish between three cases, namely, the nut fits a larger bolt, or the nut fits a smaller bolt, or the nut fits this bolt. Give a randomized algorithm which matches nuts with bolts. The expected number of comparisons (of a nut with a bolt) done by this algorithm should be $O(n \log n)$.

   Solution: This one is like randomized quick-sort. Divide the nuts and bolts into two sub-problems as follows: pick a random nut – call it n. Now find the matching bolt, call it b. Partition the bolts into two parts: those which are larger than b and those smaller than b – we can do this by comparing the bolts with n – note that we cannot just compare two bolts and say which one is larger. Similarly, by comparting with b, partition the nuts into two – those smaller than n, and those larger than n. This gives two sub-problems. As in the quick-sort algorithm, with probability at least $1/3$, both the sub-problems will have at most $2n/3$ nuts (or bolts). If this event happens, solve the two sub-problems recursively, else repeat the random selection process. The analysis is as for the quick-sort algorithm.

2. **[KT-Chapter5]** We are interested in analyzing some hard to obtain data from two databases. Each database contains n numerical values (so there are $2n$ values in total). Assume that these values are distinct. We would like to determine the median of these $2n$ values, which we define as the n^{th} smallest value. However, the only way to access these values is through queries to the databases. In a single query, we specify a value k to one of the two databases, and the chosen database returns the k^{th} smallest value that it contains. Give an algorithm which finds the median value using $O(\log n)$ queries only.

   Solution: Let the two data-bases be A and B – think of them as two arrays of size n_A and n_B respectively (initially, $n_A = n_B = n$. At any point of time, we will be interested in a sub-array of A and B. We cannot explicitly store these sub-arrays, but we just need to store the indices of the left and the right end-point of the sub-arrays. Thus, we maintain two counters l_A, r_A for A which denote the left and the right end-point of this array (initially, $l_A = 1, r_A = n$). Maintain variables l_B, r_B similarly. Now, let A' denote the part of A between l_A and r_A. Define B' similarly. First compute the median of A' and B' : we can do this by call the database with a suitable index k (for example, the median of A' will be at location l_A plus half the size of A' in the array

A. Compare the two medians, and show that you can throw away at least half of A' or B'.

3. Given an array of n objects, you need to decide if there is an object which is present more than $n/2$ times. The only operation by which you can access the objects is a function f, which given two indices i and j, outputs whether the objects at positions i and j in the array are identical or not. Given an $O(n \log n)$-time algorithm for this (where each call to f is counted as 1 operation).

   Solution: We divide the array into equal sub-arrays (left and right half). Solve the problem recursively for the two parts. Two cases happen: (i) In both the parts, there is no majority element: in this case it is easy to see that the original array does not have a majority element, (ii) In either of the two parts, the recursive call returns a majority element: let these be x and y (in case only one the recursive calls returns a majority element, then y will not be defined). Now compare x with all the elements of A to see if it appears more than half the time – if yes, x is the answer. Perform the same steps for y. If neither x nor y turn out to be a present more than $n/2$ times, then the original array cannot have a majority element.

4. (**Dasgupta, Papadimitriou, Vazirani Chapter 2**)You are given an infinite array $A[]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are not given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time.

   Solution: Start with a counter $c = 1$. We keep doubling the counter c till $A[c]$ becomes at least x – say it happens when $c = 2^i$. Then we know that c lies between $A[2^{i-1}]$ and $A[2^i]$. We also know that $2^{i-1} \geq n$. Now the size of the sub-array $A[2^{i-1}, \ldots, 2^i]$ is $2^{i-1} \leq n$. We can search for x in this sub-array using binary search – it will take $O(\log n)$ time.

5. (**Jeff Erickson**) Suppose we are given an array $A[1...n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a local minimum if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. It is possible that the array has many local minima, you are required to find any one. **Solution:** Call an array $B[1..k]$ good if $B[1] \geq B[2]$ and $B[k-1] \leq B[k]$. Note that the initial array is good. Observe that any good array must have at least one local minimum. We want to use the recurrence $T(n) = T(n/2) + O(1)$. First look at the middle two elements $x = A[n/2], y = A[n/2+1]$. Two cases arise: (i) $x \leq y$: in this case the subarray $A[1...(n/2+1)]$ is good and so we can recurse here, or (ii) $x > y$: in this case the subarray $A[n/2...n]$ is good and so we can recurse here.