

## G22.3250-001 Honors Analysis of Algorithms

**Problem 1**

An  $O(n)$  algorithm that finds the  $k$ th integer in an array  $a = (a_1, \dots, a_n)$  of  $n$  distinct integers.

**Solution (sketch):**

**BASIC IDEA USING A PIVOT:** Find an “approximate median” element  $x$  and use it as a pivot. Then partition the array  $a$  into two arrays around the pivot: the array  $a_{\text{LT}}$  of elements less than  $x$  and the array  $a_{\text{GR}}$  of elements greater than  $x$ . Let  $m$  be the size of  $a_{\text{LT}}$ . If  $k = m + 1$  then return the pivot  $x$ . If  $k \leq m$  then (recursively) find the  $k$ th element in  $a_{\text{LT}}$ , and if  $k > m + 1$  then (recursively) find the  $(k - m - 1)$ st element in  $a_{\text{GR}}$ .

**FINDING THE PIVOT:** We need to efficiently find an “approximate median”. To do so, divide the input array  $a$  consisting of  $n$  elements into  $\lceil \frac{n}{5} \rceil$  groups of (at most) 5 elements each. We then sort each group, pick out the median of each group and make these medians into a new array  $b$  of size  $\ell = \lceil \frac{n}{5} \rceil$ . We then (recursively) find the median of  $b$  using the algorithm outlined above by setting  $k = \lfloor \frac{\ell}{2} \rfloor$ .

**CORRECTNESS:** Correctness is guaranteed no matter what the quality of the approximate median! It follows simply from the parameters given in the recursion. Also the algorithm terminates since the array size decreases by at least 1 in each iteration.

**RUN-TIME ANALYSIS:** Here we analyze the quality of the “approximate median” used as a pivot. For this pivot, there are at least  $(1/2)\lceil n/5 \rceil - 2$  groups which have a larger median and each of these groups contribute at least 3 elements that are greater than the pivot (we discount the last group which might not have five elements). Hence there are *at least*  $3n/10 - 6$  elements which are larger than the pivot  $x$ . Similarly, it is easy to show that there are *at least*  $3n/10 - 6$  elements smaller than  $x$ . Hence the recursive call gets a problem instance which is of size *at most*  $7n/10 + 6$ . This means that the running time can be expressed as a recurrence relation:

$$T(n) \leq T(n/5 + 1) + T(7n/10 + 6) + O(n), \quad T(1) = O(1)$$

We can prove by induction that the above is bounded by  $T(n) \leq cn$  for some  $c$  and all large enough  $n$ . Essentially, if the inductive hypothesis holds then  $T(n) \leq c((9/10)n + 7) + c'n \leq cn + (-cn/10 + 7c + c'n)$  by setting  $n_0$  and  $c$  correctly (based on the value of  $c'$ ) we can ensure that for all  $n \geq n_0$  the latter term  $(-cn/10 + 7c + c'n)$  is not positive. □

**Problem 2**

Assuming that only equality checks are allowed, design an  $O(n)$  time algorithm to check if there is an element which occurs more than  $n/2$  times in an array containing  $n$  elements.

**Solution (sketch):**

ALGORITHM: Create an empty stack  $S$ . Go through the array one element at a time and, for each new element  $e$ : (1) if the stack is empty push  $e$  on the stack  $S$  (2) if the top of the stack  $e'$  is equal to  $e$  then push  $e$  on the stack (3) otherwise pop the element  $e'$  which is at the top of the stack  $S$  and ignore  $e$ .

Once this is done, if  $S$  is non-empty then let  $e$  be the element at the top of  $S$ . count the number of occurrences of  $e$  in the original array. If there are more than  $n/2$  of them, return "true", otherwise return "false".

CORRECTNESS: Firstly, we notice that any given point, all elements in  $S$  have the same value. This is because we only push an element onto the stack if it is equivalent to the top of the stack. Secondly, any element that does not end up in the stack must have either been ignored or popped in condition (3) above. We can uniquely pair up all such elements into disjoint pairs  $(e, e')$  (where  $e$  is the element that was ignored in (3) and  $e'$  is the one that got popped) so that  $e \neq e'$ . Lastly, if some value  $v$  occurs more than  $n/2$  times, then it is impossible that *all* occurrences of  $v$  are paired up as above. Hence some element whose value is  $v$  must end up in the stack. Since we saw all elements in the stack have the same value, the top of the stack must have value  $v$ . This concludes the proof of correctness. (Note that the implication only goes in one direction and we cannot say that if some value ends up in the stack it occurs more than  $n/2$  times. Therefore we need the final check).

RUN TIME: We do at most two iterations over the array and hence it is clear that the run-time is  $O(n)$

□

### Problem 3

Suppose  $a > b > 0$  and  $c > 0$  are constants and

$$T(n) = aT\left(\frac{n}{b}\right) + cn, \quad T(a) \leq c$$

Show that  $T(n) = O(n^{\log_b a})$ .

**Solution (sketch):** One way to solve the problem is to look at the recursion tree. Think of the above recurrence relation as being for a recursive algorithm which makes  $a$  recursive calls, in each of which the problem size is  $n/b$ . In addition, it does  $cn$  work aside from the recursive calls. Then let us look at tree of problems being solved:

Level	Problem Size	Number of Problems	Work Per Problem	Total Work for Level
0	$n$	1	$cn$	$cn$
1	$n/b$	$a$	$c(n/b)$	$ac(n/b)$
2	$n/b^2$	$a^2$	$c(n/b^2)$	$a^2c(n/b^2)$
i	$n/b^i$	$a^i$	$c(n/b^i)$	$a^i c(n/b^i)$

Hence the total amount of work is  $T(n) = cn \sum_{i=1}^m \left(\frac{a}{b}\right)^i$  where,  $m = \log_b(n)$  is the number of levels (since the problem size at level  $m$  is  $n/b^m = 1$ ).

Now we solve

$$\sum_{i=1}^m \left(\frac{a}{b}\right)^i = \frac{(a/b)^{m+1} - 1}{a/b - 1} \leq \left(\frac{a/b}{a/b - 1}\right) \left(\frac{a^{\log_b(n)}}{n}\right) = c'n^{\log_b(a)-1}$$

For some constant  $c'$ . Plugging this in, we get  $T(n) = (cn)c'n^{\log_b(a)-1} = c''n^{\log_b(a)} = O(n^{\log_b(a)})$  as we wanted to show. □

## Problem 4

Given  $n$  intervals of real numbers,  $[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]$  design an  $O(n \log n)$  time algorithm to decide whether there exists a pair of intervals that overlap.

**Solution (sketch):** Sort the intervals by their starting point  $a_i$  resulting in a sorted list  $[a_{\pi(1)}, b_{\pi(1)}], \dots, [a_{\pi(n)}, b_{\pi(n)}]$  for some permutation  $\pi$ . Then scan the successive sorted intervals to see if  $a_{\pi(i+1)} \leq b_{\pi(i)}$  for any  $i$  and, if so, there is an overlap. Otherwise it could be argued there is no overlap.

The algorithm only involves sorting and one scan so the run time (assuming a good sorting algorithm) is  $O(n \log n)$ . □

## Problem 5

Given a  $m \times n$  matrix of integers such that every row is strictly increasing (from left to right), and every column is strictly increasing (from top to bottom), design an  $O(m + n)$  time algorithm to test if a given integer  $b$  is contained in the matrix.

**Solution (sketch):** The algorithm starts with the matrix  $M$  and the indices  $i = m, j = 1$  (corresponding to the bottom left of the matrix) and in each iteration: (1) if  $M[i, j] = b$  return true, (2) if  $M[i, j] < b$  then set  $j := j + 1$  (go right) (3) if  $M[i, j] > b$  then  $i := i - 1$  (go up). If  $i$  or  $j$  “falls off” the matrix ( $i = 0$  or  $j = n + 1$ ) then return “false”.

For correctness, we can think of  $M[i, j]$  as a pivot and, in each iteration we have the invariant that  $b$  must be to the top and right of the pivot (or the pivot itself). This is true at the beginning and, each time we go up or right, we discard a row or a column which cannot contain  $b$ .

It is easy to see that the run-time is  $O(n + m)$  since, after  $n + m$  iterations, one of  $i, j$  must “fall off” the matrix. □

## Problem 6

Given a sequence of positive integers  $(a_1, a_2, \dots, a_n)$ , design an  $O(n)$  time algorithm to find a shortest sub-sequence of consecutive integers  $(a_i, a_{i+1}, \dots, a_j)$  whose sum is at least a given integer  $M$ .

**Solution (*sketch*):** We use a “sliding window” to find this subsequence. Start of with two indices  $i = 1, j = 1$ . In each iteration, we first do an *increase* step where we increase  $j$  until the sequence  $a_i, \dots, a_j$  adds up to  $M$ . Then we do a *narrow* step in which we increase  $i$  until we get the largest value for which  $a_i, \dots, a_j$  adds up to  $M$ . At the conclusion of these steps we have a *candidate window*  $(i, j)$ . If it smaller than the *current best* (or there is no current best yet) we set  $(i, j)$  as the current best. We set  $j := j + 1$  and continue with the next iteration (until  $j$  “falls off” the array). We return the *current best solution* at the end.

It is clear that the run time is  $O(n)$  since, in each step, we either increase  $i$  or  $j$ .

For correctness, we need to argue that, if  $(a_{i_0}, \dots, a_{j_0})$  is the shortest subsequence then, at some point in the algorithm, we get the candidate window  $(i_0, j_0)$ . To see this, we consider the first time that the index  $i$  reaches  $i_0$ . At the beginning of this iteration,  $j$  must have reached exactly  $j_0$  (if it was less then  $j_0$  then  $i$  cannot move up to  $i_0$  and  $j$  cannot go beyond  $j_0$  since this already adds up to  $M$ ). Hence we consider the window  $(i_0, j_0)$  and we will find a sequence that is at least as short.  $\square$