

**Question 1.** To solve  $\text{Median}(S)$  using  $\text{Select}(S, k)$  is trivial: Simply call  $\text{Select}(S, |S|/2)$  and return its answer as the answer to  $\text{Median}(S)$ . To solve  $\text{Select}(S, k)$  using  $\text{Median}(S)$  first call  $\text{Median}(S)$  and let  $x$  be the element it returns. Let  $S_{<}$  denote the subset of  $S$  consisting of those elements of  $S$  that are  $< x$ , and let  $S_{>}$  denote the set consisting of those elements of  $S$  that are  $> x$ ; obviously  $S_{\leq}$  and  $S_{>}$  can be computed in  $O(|S|)$  time (by going through  $S$  once). If  $k = |S|/2$  then we return  $x$  as the answer. If  $k < |S|/2$  we recursively call  $\text{Select}(S_{<}, k)$  and return its answer. Otherwise (i.e., if  $k > |S|/2$ ) we recursively call  $\text{Select}(S_{>}, k - 1 - |S|/2)$  and return its answer. A linear-time algorithm for  $\text{Median}(S)$  implies that the recurrence for the time of  $\text{Select}(S, k)$  is

$$T(n) = c_0 \text{ if } n \leq \text{some constant } n_0$$

$$T(n) \leq T(n/2) + cn \text{ if } n > n_0.$$

Since  $0.5 < 1$ , the solution to this recurrence is  $T(n) = O(n)$  (as explained in class).

**Question 2.** The main idea is to compare the median element of  $A$  to the median element of  $B$ : If the former is greater, then the elements in the larger half of  $A$  can be thrown away (for being too big – their ranks exceed  $n$ ), and the elements in the smaller half of  $B$  can also be thrown away (for being too small – their ranks are surely smaller than  $n$ ); in other words we recurse on the smaller half of  $A$  and the larger half of  $B$ . If it is the median of  $B$  that is greater than the median of  $A$ , then we simply interchange the roles of  $A$  and  $B$  in the previous sentence. The reason it is correct to recurse in this way is that the element we are searching for has, in the recursive call, “lost”  $n/2$  elements smaller than it and  $n/2$  elements greater than it (the halves of  $A$  and  $B$  that were “thrown away”), and therefore its rank in the reduced-size problem is  $n - (n/2) = n/2$ . A more formal exposition is given below.

To simplify the exposition, assume that  $A[1] < A[2] < \dots < A[n]$  and  $B[1] < B[2] < \dots < B[n]$ , that  $n$  is a power of 2, and that the elements of  $A$  and  $B$  are distinct, i.e.,  $A[i] \neq B[j]$  for all  $i, j$ . This question is about designing an efficient algorithm for finding the  $n$ -th smallest of the  $2n$  combined elements of  $A$  and  $B$ .

In the recursive algorithm given below, we use the notation  $A[i : j]$  to denote the subarray of  $A$  from positions  $i$  to  $j$  (with  $i \leq j$ ). The algorithm maintains the invariant that the sizes of the relevant subarrays are equal, and are powers of two (i.e.,  $j - i + 1 = l - k + 1 =$  a power of two). The algorithm is supposed to return the  $(j - i + 1)$ th smallest element in the union of  $A[i : j]$  and  $B[k : l]$ ; for computing the  $n$ -th smallest of the  $2n$  combined elements of  $A$  and  $B$ , it is simply called with  $\text{Select}(A[1 : n], B[1 : n])$ .

**Algorithm**  $\text{Select}(A[i : j], B[k : l])$

1. If  $i = j$  (hence  $k = l$ ) then return  $\min\{A[i], B[k]\}$ , otherwise continue to the next step.
2. Compare  $A[(j + i - 1)/2]$  to  $B[(l + k - 1)/2]$ . Assume without loss of generality that  $A[(j + i - 1)/2] > B[(l + k - 1)/2]$  (otherwise interchange the roles of  $A$  and  $B$  in what follows).

3. Return **Select**( $A[i' : j']$ ,  $B[k' : l']$ ) where  
 $i' = 1 + (j + i - 1)/2$ ,  $j' = j$ ,  $k' = k$ ,  $l' = (l + k - 1)/2$ .

The algorithm's time complexity satisfies the recurrence:

$$T(1) = c \text{ where } c \text{ is a constant}$$

and, for  $n > 1$ :

$$T(n) = T(n/2) + d \text{ where } d \text{ is a constant.}$$

The above recurrence is similar to the one for binary search, and has a solution  $T(n) = O(\log n)$ .

**Question 3.** Let  $S_i = \{I_1, \dots, I_i\}$ , i.e.,  $S_i$  is the subset of  $S$  none of whose intervals extends to the right of  $r_i$ . Let  $C_i$  be a maximum-weight acceptable subset of  $S_i$  that contains interval  $I_i$ ; in other words the intervals of  $C_i$  have maximum total weight subject to the constraints of being nonoverlapping and including  $I_i$ . Note that if we had all of  $\{C_1, C_2, \dots, C_n\}$  then we could compute the answer in linear time by simply choosing the largest of the  $C_i$ s. Hence it suffices to compute all the  $C_i$ s.

Observe that  $C_1 = w_1$  and that, for  $i > 1$ , we have

$$C_i = w_i + \max_{k: r_k < l_i} C_k$$

which immediately implies an  $O(n^2)$  time algorithm for computing all the  $C_i$ s (by computing each of  $C_1, C_2, \dots, C_n$  in that order, according to the above equation).

**Question 4.** For  $i = 0, 1, \dots, (n/k) - 2$ , let  $S_i$  denote the subarray of  $A$  that has length  $2k$  and begins at the  $(ik + 1)$ th position in  $A$ . The algorithm consists of sorting (in that order) the subarray of  $A$  called  $S_0$ , then the subarray called  $S_1$  (whose contents have of course changed as a result of sorting  $S_0$ ), ..., then the portion called  $S_{(n/k)-2}$ . Correctness of this is seen by observing that, after sorting  $S_0$ , the leftmost  $k$  items of  $A$  are at their final position in the sorted version of  $A$  (this follows from the property). The same is true for the leftmost  $k$  items of each  $S_i$  after we are done sorting it. The time for sorting each  $S_i$  is  $O(k \log k)$ , and since this must be done  $(n/k) - 1$  times the overall time is  $O(n \log k)$ .