Sample Solution to Problem Set 1

2. (10 points) Selection of multiple keys

In class, we studied the problem of selection of an element of a given rank from an unsorted list. In this problem, we consider the problem of simultaneously selecting keys corresponding to multiple (say k) ranks from an unsorted list of n elements. It is not hard to see that one can solve the above problem using the single selection algorithm in $O(nk)$ time. Can we do better?

Formally, the input to the problem is a set S of n distinct keys drawn from some totally ordered universe, and a set of $k+1$ integers r_i such that $1 = r_0 < r_1 < \cdots < r_k = n+1$. The output is a partition of S into k sets $S_0, \ldots, S_{k-1}$ such that S_i is the set of all keys with ranks greater than or equal to r_i and strictly less than r_{i+1}.

(a) Give an $O(n \log k)$-comparison divide and conquer algorithm for the problem.

Answer: We give a divide-and-conquer algorithm for this *multi-selection* problem. We find the element of rank $r_{\lceil k/2 \rceil}$ in S using SELECT in $O(n)$ time. We next partition the set S into two groups around this element, while keeping the element of rank $r_{\lceil k/2}$ in the right group. We then recursively call the multi-selection procedure on the two groups, the left one with the ranks $r_0, r_1, \ldots, r_{\lceil k/2 \rceil - 1}, r_{\lceil k/2 \rceil}$, and the right one with the ranks $1, r_{\lceil k/2 \rceil + 1} - r_{\lceil k/2 \rceil} + 1, \ldots, r_k - r_{\lceil k/2 \rceil} + 1$.

The correctness of the above procedure follows from the correctness of the select and partition procedures. A formal proof, by induction on k, is left to the reader.

We get the following recurrence for the running time:

$$T(n,k) \leq \Theta(n) + T(m, k/2) + T(n-m, k/2).$$

The above recurrence is written as a two-variable recurrence, but it is easy to solve using the recursion tree approach. We note that the contribution of each level is $\Theta(n)$, and the number of levels is $\Theta(\lg k)$ since the number of keys goes down by half in each level. This establishes that the running time of the above multi-selection algorithm is $\Theta(n \lg k)$.

In the above recurrence, we assumed that k was a power of 2. If not, then we get a recurrence of the form

$$T(n,k) \leq \Theta(n) + T(m, \lfloor k/2 \rfloor) + T(n-m, \lfloor k/2 \rfloor).$$

We can establish by induction on k that $T(n,k) \leq cn \lg k$ for $k \geq 2$, and $T(n,k) \leq cn$ for $k = 1$, where $c > 0$ is a constant suitably chosen. We sketch the proof here. For the base case, we consider $k \in \{1,2,3,4\}$. For $k = 1$, we have a call to the Select procedure, which costs at most cn time. We can analyze for $k = 2$, 3, and 4 using a simple argument. For

the induction hypothesis, we assume the claim holds for $k < \ell$. For $k = \ell$, we invoke the induction hypothesis for $\lfloor \ell/2 \rfloor$ (which is at least 2) to obtain

$$\begin{aligned} T(n,\ell) &\leq c_1 n + cm\lg(\ell) - cm + c(n-m)\lg\ell - c(n-m) \\ &\leq c_1 n + cn\lg\ell - cn \\ &\leq cn\lg\ell, \end{aligned}$$

by selecting c suitably bigger than c_1. Here c_1 is the constant hidden in the Θ notation of the recurrence above.

(b) Show that in the comparison based model, every algorithm incurs $\Omega(n \log k)$ comparisons on its worst-case instance. (*Hint:* Use the approach used for placing a lower bound on the number of comparisons needed for sorting. See Section 8.1 of text.)

Answer: We use the decision-tree method to establish a lower bound for the multi-selection problem. Any comparison-based multi-selection algorithm can be written as a decision-tree in which each node represents a comparison.

Let $\mathcal{A}$ be any comparison-based algorithm for multi-selection over n elements and k ranks and let h and N be the height and the number of nodes the decision-tree associated with $\mathcal{A}$. The number of leaf nodes should be at least the number of different output configurations. The number of different output configurations equals

$$\binom{n}{r_1}\binom{n-r_1}{r_2-r_1}\binom{n-r_2}{r_3-r_2}\cdots = \frac{n!}{r_1!(r_2-r_1)!(r_3-r_2)!\cdots 1}.$$

Consider the specific instance in which $r_i = in/k$. In this case, the right hand side of the above equality reduces to $n!/((n/k)!)^k$. Therefore, the height of the tree h must satisfy:

$$\begin{aligned} h &\geq \lg(n!) - k\lg((n/k)!) \\ &= n\lg n - \Theta(n) - k(n\lg(n/k)/k - \Theta(n/k)) \\ &= n\lg n - n\lg n + n\lg k + \Theta(n) \\ &= \Omega(n\lg k). \end{aligned}$$

Since the height of the tree is a lower bound on the number of comparisons needed in the worst-case, the desired claim is proved.

3. (10 points) Selection from two databases

You are interested in selecting elements of a given rank from the union of two separate databases. One database has m numbers and the other has n numbers, and you may assume that all the values are different. The only access you have to each database is through a query where you specify a rank k, and the chosen database returns the kth smallest element that it contains.

Give an algorithm that finds an element of a given rank k from the union of the two databases, while making $O(\log(\min\{m,n\}))$ queries.

Answer: We represent the two databases as two sorted lists $A[1..m]$ and $B[1..n]$. Without loss of generality, assume that $n \leq m$. The worst-case running time of our algorithm will turn out to be $O(\log n)$; thus, in general, $O(\log(\min\{m,n\}))$.

The basic idea is to reduce the size of array B by half in each iteration by performing only a consant number of operations. Thus, the number of iterations will be $O(\log n)$ and so will the worst-case running time be. In the following pseudocode, A and B are the two given arrays, p and r are two indices of array B, $p \leq r$, and k is the rank that we are seeking. The algorithm returns the element of rank k from the two sorted lists $A[1..m]$ and $B[p..r]$. For convenience, we assume that $A[0] = -\infty$.

$2ListSelect(A, B, p, r, k)$

0. **if** $r < p$ **return** $A[k]$ // B is empty
1. **if** $k < r - p + 1$ // No point in searching $B[p + k..r]$
2. $r = p + k - 1$
3. **if** $p = r$ // B has only one element
4. **if** $A[k-1] > B[p]$ **return** $A[k-1]$
5. **else if** $A[k-1] < B[p]$ **return** $\min\{A[k], B[p]\}$
6. **else** // B has more than one element
7. $\ell = \lfloor (r - p + 1)/2 \rfloor$ // ℓ is half the size of B
8. **if** $A[k - \ell] < B[p + \ell - 1]$ // compare the middle element of B with appropriate element of A
9. **return** $2ListSelect(A, B, p, p + \ell - 1, k)$ // discard top half of B
10. **else return** $2ListSelect(A, B, p + \ell, r, k - \ell)$ // discard bottom half of B

We prove the correctness of $2ListSelect(A, B, p, r, k)$ by induction on the size of list B, which is $r - p + 1$. The base case is when $r - p + 1 = 0$. In this case B is empty and the algorithm returns $A[k]$; so the base case is established.

We now consider the induction step. Let the length of B be $x = r - p + 1 > 0$. For the induction hypothesis, we assume that the algorithm is correct for lengths less than x. If $k < x$, then we truncate the list B by setting the upper index to $p + k - 1$, thus setting the length of B to k. Clearly, all the elements in $B[p + k..r]$ are of rank larger than k in $A \cup B[p..r]$. So an element of rank k is in $A \cup B[p..p + k - 1]$. So, after truncation and setting r appropriately (if needed), we need to determine the element of rank k in $A \cup B[p..r]$. If $B[p..r]$ has only one element, then steps 4 and 5 ensure that the correct element among $\{A[k-1], A[k], B[p]\}$ is returned.

Now consider the case when $B[p..r]$ has more than one element. We compare $A[k-\ell]$ with $B[p+\ell-1]$ in line 8, where $\ell = \lfloor (r - p + 1)/2 \rfloor$ is at most half the size of B. Note that there are exactly k elements in $A[1..k - \ell] \cup B[p..p + \ell - 1]$. If $A[k - \ell] < B[p + \ell - 1]$, then there are at least $k - \ell$ elements in A and ℓ elements in B that are smaller than each element in $B[p + \ell..r]$; thus, we can eliminate the top half of $B[p..r]$ and recurse on A and $B[p..p+\ell-1]$, which is what we do in line 9. Otherwise, the only elements that can be at most any element of $B[p..p + \ell - 1]$ are the elements in $B[p..p+\ell-1]$ and the elements of $A[1..k-\ell-1]$, which number to $k-1$; thus, we can eliminate the bottom half of B and recurse on $A[1..m]$ and $B[p + \ell..r]$ with the desired rank set to $k - \ell$, as done in line 10. The value returned by our recursion is correct by the induction hypothesis. This completes the proof of the induction step.

In recursive step, we decrease the length of B by a factor of 2 after performing only a constant number of probes and other operations. So the worst-case running time of the algorithm is given by the recurrence $T(n) = T(\lceil n/2 \rceil) + \Theta(1)$, which yields $T(n) = \Theta(\log n)$ (like binary search).

4. (10 points) Finding a local maximum in a grid

You are given an $n \times n$ grid, each cell of which contains a distinct integer. Each cell can be identified by the pair (r, c), $1 \leq r, c \leq n$, where r gives the row number and c gives the column number. Your access to grid is through a *probe*, by which you determine the integer stored in a given cell.

We say that (i, j) is a *neighbor* of a given cell (r, c) if and only if $|i - r| + |j - c| = 1$. We say that a cell (r, c) is a *local maximum* if the integer at (r, c) is greater than the integers stored at each of its neighboring cells.

Give an algorithm that determines a local maximum in a grid using $O(n)$ probes.

Answer: We present a divide and conquer algorithm. In the divide step, we will first check if the border of the given $n \times n$ grid has a local maximum. If so, then we simply return it and are done. Otherwise, we identify a quadrant of the grid, that is, an $n/2 \times n/2$ sub-grid which has a local maximum, and recurse in that subgrid.

We now fill the details. First, if n is 1, then we simply return the lone element. Otherwise, we consider two cases.

1. In the first case, there is a local maximum in the "border" B of the grid. We check this by processing all the items in rows 1 and n, and columns 1 and n, and check if there is any element that is larger than each of its neighbors. There are at most $4n$ elements in the border, so this process takes $O(n)$ time. If a local maximum is detected, then we return that element as a local maximum.

2. The remaining case is when we do not find a local maximum in the border. In this case, we will reduce the problem to a quarter of the grid so that, we recurse from an $n \times n$ grid to an $n/2 \times n/2$ grid.

   (a) Let C denote the middle column, and R denote the middle row.

   (b) As in the step above, we check if there is a local maximum in $C \cup R$. Again, since $C \cup R$ has $O(n)$ elements, this takes $O(n)$ time. If we find a local maximum, then we return it and are done.

   (c) Otherwise, we are in a situation where there is no local maximum in $B \cup C \cup R$. Let u be the largest element in all of $B \cup C \cup R$. Since u is not a local maximum, one of its neighbors, say v, in one of the quadrants, say Q, is even larger. We note that Q has a local maximum: the global maximum inside Q is in the interior of Q, not in its border since v is larger than every element in the border of Q. Therefore, the global maximum inside Q is, in fact, a local maximum in Q and of the whole grid.

   So we make a recursive call for finding a local maximum in Q, and return its local maximum.

The running time of the algorithm is given by the recurrence.

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

since each of the steps other than the recursive call takes $O(n)$ time, and the recursive call is to a grid of size $\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor$. We can rewrite the above recurrence as follows (c is an appropriate

constant).

$$\begin{aligned} T(n) &\leq cn + T(n/2) \\ &\leq c(n + n/2) + T(n/4) \\ &\leq c(n + n/2 + \ldots 2) + T(1) \\ &= 2cn \end{aligned}$$

So we obtain $T(n) = O(n)$.