Introduction to Algorithms	**Solution Sketches for Problem Set 5**
CS 4820, Spring 2018	**Due 11:59pm Thursday, March 8**

(1) *(Note: This problem is Exercise 5.3 in Kleinberg & Tardos, with an extra "part b" appended.) Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are* equivalent *if they correspond to the same account.*

It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them into the equivalence tester.

(a) (5 points) *Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.*

(b) (5 points) *Now modify the question so that you must decide whether there is a set of more than $n/4$ of the cards that are all equivalent to one another. Show how to decide the answer to this question with only $O(n \log n)$ invocations of the equivalence tester.*

If you are confident that you have a correct solution to part (b) of this problem, you can skip part (a) and your score on the problem will be computed by doubling your score for part (b).

Solution. Here's a solution that covers both parts, by solving the following more general problem: *for any specified number $k \geq 1$, using $O(kn \log(n/k))$ equivalence tests, output one representative of each equivalence class that contains more than n/k cards.*

The algorithm is recursive. If there are fewer than k cards, output all of them. Otherwise, partition the cards arbitrarily into two sets, which we'll call the "left" and "right" sets, each of size at most $\lceil n/2 \rceil$. Run the algorithm recursively on each piece of the partition. Let L and R be the outputs of running the algorithm on the left and right sets, respectively. For each card in $L \cup R$, do an equivalence test of that card versus each of the $n-1$ other cards. Find all the cards in $L \cup R$ whose equivalence class contains more than n/k cards, group them into equivalence classes, and output one element of each equivalence class.

To prove correctness, observe that we never output a card unless we have directly equivalence-tested it against all $n-1$ other cards and found that its equivalence class contains more than n/k cards. So our algorithm produces no false positives. We claim that there are also no false negatives, i.e. if an equivalence class C contains more than n/k cards, then our algorithm will definitely output an element of C. To prove this, we use strong induction on the number of cards. The base case, when there are fewer than k cards, is trivial since we output every card in that case. For the induction step, let S_L and S_R denote the left and right sets. The relations

$$|C| > n/k, \quad |C| = |C \cap S_L| + |C \cap S_R|, \quad n = |S_L| + |S_R|$$

collectively imply that at least one of $|C \cap S_L| > |S_L|/k$ or $|C \cap S_R| > |S_R|/k$ holds true. By the induction hypothesis, either L or R will contain an element of C. Therefore in the final phase of the

algorithm we will equivalence-test this element against every other card and we'll detect that $|C| > n/k$ and output the element.

To judge the worst-case number of equivalence tests, note that the number of equivalence classes with more than n/k elements is always less than k. So the two recursive calls each produce fewer than k cards, meaning that $L \cup R$ has fewer than $2k$ elements. Each of these is equivalence-tested against fewer than n alternatives. Hence, if $T(n)$ denotes the worst-case number of equivalence tests performed by the algorithm when the set contains n cards, then

$$T(n) \leq \begin{cases} 0 & \text{if } n < k \\ 2T(n/2) + 2kn & \text{otherwise.} \end{cases}$$

The solution to this recurrence is $T(n) = O(kn \log(n/k))$.

Alternate solution 1 for part (a). Group the elements in pairs. Test each pair for equivalence. If they are inequivalent, throw them both away. Retain one element from each of the remaining pairs; let r denote the number of elements thus retained. Recursively find the majority equivalence class (if there is one) among those r elements. Let x be a representative of this equivalence class. Test x for equivalence with each of the original elements of the original n-element set. If it matches at least $(n-1)/2$ of them then you've found a majority equivalence class, otherwise there is none.

This algorithm isn't quite right, because it doesn't say what to do when n is odd. If n is odd then in the original grouping there is one singleton, and this singleton becomes one of the r retained elements no matter what.

The running time satisfies $T(n) = O(n) + T(\lceil n/2 \rceil)$, which solves to $T(n) = O(n)$. To prove correctness, notice once again that the algorithm can't produce false positives. However, if there is a majority equivalence class C then when we group the elements into pairs and throw away the pairs where the elements don't match each other, we're throwing away at most one element of C and at least one element of the complement of C each time. Thus, C is still a majority equivalence class among the $2k$ remaining elements. When we cut this down to k elements by picking one from each matched pair, we are throwing away exactly as many elements of C as its complement, so once again C remains the majority equivalence class. At that point, we apply the induction hypothesis to assert that x will be an element of C, and then it's easy to see that the algorithm will discover that C is the majority equivalence class.

Alternate solution 2 for part (a). Order the elements arbitrarily from 1 to n. Initialize a stack with the first element. As you go through elements 2 through n, compare them with the element at the top of the stack. If they are equivalent, push the new element onto the top of the stack. If they aren't equivalent, pop the stack and throw away both elements. At the end, if the stack is non-empty, take the element x from the top of the stack. Test x for equivalence with each of the original elements of the original n-element set. If it matches at least $(n-1)/2$ of them then you've found a majority equivalence class, otherwise there is none.

The algorithm clearly makes $O(n)$ equivalence tests. There's nothing divide-and-conquer about it. For correctness, as always there can be no false positives. If C is a majority equivalence class, consider that each time you throw away an element of C you are also throwing away an element of its complement. Therefore, you can't possibly throw away every element of C which means that the stack contains at least one element of C at the end. All the elements in the stack are equivalent to each other (because each is equivalent to the one immediately below it) and therefore they all belong to C. So x belongs to C and in the final step we'll detect that C is a majority equivalence class.

(2) *(10 points) Solve Exercise 5.7 in Kleinberg & Tardos.*

Solution. We begin with the following observation: if a grid graph contains an interior node at which the value is less than the minimum value on the boundary, then there is an interior node which is a local minimum. The proof of the observation is almost trivial: the global minimum value cannot lie on the boundary, so it lies in the interior, and the node containing the global minimum value is obviously a local minimum.

So here's the divide and conquer algorithm. The algorithm is designed to search a rectangular grid whose horizontal and vertical dimensions differ by at most 1, i.e. the grid is either an n-by-n square or an n-by-$(n \pm 1)$ rectangle. In the base case, if we have a grid of dimension 2-by-2 or smaller, simply probe every grid cell and output the global minimum. Otherwise partition the n-by-n grid into four sub-grids whose horizontal and vertical dimensions are each in the range $\{\lfloor n/2 \rfloor, \lceil n/2 \rceil\}$. Let S be the set of nodes in the union of the boundaries of the four sub-grids. Probe every node in S, and let v denote the one whose label is minimum. If the value at v is less than the four neighbors' values, then v is a local minimum; output it. Otherwise v has a neighbor (lying in the interior of one of the sub-grids) whose label is even smaller. Recurse on that sub-grid.

The set S has size $O(n)$, so the number of probes performed by the algorithm satisfies $T(n) = T(\lceil n/2 \rceil) + O(n)$, whose solution is $T(n) = O(n)$.

The proof of correctness is by strong induction. The induction hypothesis is: *whenever we run this algorithm on a grid whose larger dimension is n, it outputs a local minimum whose value is less than or equal to the minimum value attained on the boundary of the grid.* The base case $n \leq 2$ is trivial because we always output a global minimum. For the induction step, we assume the induction hypothesis is true for every grid whose larger dimension is strictly less than n, which includes the four sub-grids. If the node v defined in the algorithm above is a local minimum, then it is clearly correct to output v. Otherwise, the induction hypothesis ensures that we output a node w which is a local minimum of the sub-grid, and that $x_w \leq x_v$. This local minimum of the sub-grid must be a local minimum of the full grid: if w has any neighbor u in the full grid which doesn't belong to the sub-grid, then $u \in S$ which implies that $x_u \geq x_v \geq x_w$. Also, the value x_w is less than or equal to the minimum value attained on the boundary of the grid, because if u is a node on the boundary of the grid then $u \in S$ which again implies $x_u \geq x_v \geq x_w$ as above. Thus, we have confirmed the induction step, and the correctness of the algorithm follows.

(3) *(10 points)* A closed axis-parallel rectangle in the plane is a subset $R \subset \mathbb{R}^2$ that is the Cartesian product of two closed intervals,

$$R = [a, b] \times [c, d] = \{(x, y) \mid a \leq x \leq b \text{ and } c \leq y \leq d\}.$$

We call the 4-tuple (a, b, c, d) the *description* of R. Given the descriptions of n rectangles, design an algorithm to decide whether there exists a point in the plane that belongs to two or more of the rectangles. Your algorithm's running time should be $O(n \log n)$.

Solution. We will present a solution for the special case in which the numbers $a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n$ are all distinct. Our algorithm simulates a vertical scan line moving across the set of rectangles from left to right. The intersection of each rectangle R_i with the vertical line $\{x = c\}$ is a (possibly empty) interval. We will maintain a data structure storing these intervals, in a way that lets us efficiently test if two of the intervals intersect.

As a preprocessing step, sort the union of the sets $\{a_1, a_2, \ldots, a_n\} \cup \{b_1, b_2, \ldots, b_n\}$ in increasing order. These are all the x-coordinates at which the set of rectangles that intersect the scan line changes. Let $t_1, t_2, \ldots, t_{2n}$ denote the elements of this sorted set. We'll call a rectangle "active at time t" if $a_i \leq t \leq b_i$.

Initialize a balanced binary search tree (e.g. a red-black tree) that supports insertion, deletion, and finding the predecessor and successor of an element in $O(\log n)$ time. This search tree will store the

set of endpoints of the intervals $[c_i, d_i]$ of all the active rectangles. We initialize the search tree to be empty and we run a sequence of $2n$ iterations corresponding to the elements of the set $t_1, t_2, \ldots, t_{2n}$. In the iteration corresponding to t_j, if $t_j = b_i$ for some rectangle R_i then we delete c_i and d_i from the search tree. If $t_j = a_i$ for some rectangle R_i then we insert c_i and d_i into the search tree and perform an "intersection check" that tests whether $[c_i, d_i]$ intersects any of the intervals represented by the other active rectangles. The intersection check outputs "yes" if either c_i, d_i are inserted into non-consecutive locations in the search tree, or if c_i, d_i are consecutive and the element immediately preceding them is equal to c_k for some other rectangle R_k.

The algorithm performs an $O(n \log n)$ preprocessing step to generate the sorted list $t_1, \ldots, t_{2n}$, and after that it performs a sequence of $O(n)$ operations, each of which takes $O(\log n)$ time because the balanced binary search tree supports insertion, deletion, predecessor, and successor in $O(\log n)$. (Actually, the algorithm as described above never uses the successor operations, only the predecessor.) Hence the total running time is $O(n \log n)$.

We only sketch the proof of correctness. First one proves a lemma saying that after the end of the iteration corresponding to t_j, the set of elements of our binary search tree consists of all rectangles that intersect the line $\{x = c\}$ for all c in the open interval (t_j, t_{j+1}). The proof is by induction on j. Next one proves a second lemma saying that, assuming the first lemma holds, the intersection check is a valid procedure to test whether two of the rectangles intersect at a point whose x-coordinate belongs to the interval (t_j, t_{j+1}). Finally, using the assumption that $a_1, \ldots, a_n, b_1, \ldots, b_n$ are all distinct, one proves that if there is an intersecting pair of rectangles, then there is an intersection point whose x-coordinate belongs to one of the half-open intervals (t_j, t_{j+1}) where $t_j \in \{a_1, a_2, \ldots, a_n\}$.