Introduction to Algorithms **Solution Set 3**
CS 482, Spring 2008

(1) We describe an algorithm TEST(S) whose input is a set S of bank cards and whose output is:

- a bank card $x \in S$ such that more than half the elements of S are equivalent to x, if any such bank card exists;
- otherwise, the output is NULL.

Let $n = |S|$. When $n = 0$, TEST(S) outputs NULL. When $n = 1$, TEST(S) outputs the unique element of S. Otherwise, TEST(S) partitions S into two (arbitrary) subsets S_1, S_2 of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Let $x_1 = \text{TEST}(S_1), x_2 = \text{TEST}(S_2)$. For each non-NULL element of the set $\{x_1, x_2\}$, the algorithm takes this bank card and performs an equivalence test against all other bank cards. Let n_1 be the number of bank cards equivalent to x_1 (including x_1 itself), or $n_1 = 0$ if $x_1 = \text{NULL}$. Define n_2 similarly, but with x_2 in place of x_1. If $n_1 > n/2$ then TEST(S) outputs x_1; else if $n_2 > n/2$ then TEST(S) outputs x_2; else TEST(S) outputs NULL.

Running time. Equivalence-testing x_1 against all other bank cards takes $n - 1$ comparisons or 0 comparisons, depending on whether $x_1 = \text{NULL}$, and similarly for x_2. Thus the running time, $f(n)$, of TEST(S) satisfies

$$f(n) \leq f\left(\lfloor n/2 \rfloor\right) + f\left(\lceil n/2 \rceil\right) + 2(n-1).$$

The solution of this recurrence satisfies $f(n) = O(n \log n)$.

Correctness. The proof of correctness is by induction on n, the base cases $n = 0, 1$ being trivial. The algorithm never outputs a bank card unless it has directly compared that bank card to all others and determined that it is equivalent to at least $\lfloor n/2 \rfloor$ of them, hence it never produces an incorrect non-NULL output. We only need to prove that if there is a set $T \subseteq S$ of bank cards which are all equivalent, such that $|T| > n/2$, then the algorithm outputs an element of T. To prove this, let

$$m = |T|, \quad m_1 = |T \cap S_1|, \quad m_2 = |T \cap S_2|.$$

If $2m_1 \leq |S_1|$ and $2m_2 \leq |S_2|$ then $2m \leq |S|$ contradicting our assumption that $|T| > n/2$. So at least one of the following inequalities holds:

$$m_1 > |S_1|/2 \quad \text{or} \quad m_2 > |S_2|/2.$$

Assume without loss of generality that the first of these holds. Then TEST(S_1) outputs an element $x_1 \in T$, and TEST(S) will compare x_1 with all other elements of S and discover that the set of elements equivalent to x_1 (i.e., the set T) contains more than half the elements of S. Hence TEST(S) will output an element of T, as desired.

(2) We present two solutions. The first is a divide-and-conquer algorithm which resembles mergesort in that it divides the input into two equal-sized subproblems, solves both subproblems, then combines the results by carefully walking through a pair of ordered lists. The second is a simpler algorithm that sorts the lines and processes them in order of increasing slope, maintaining a stack whose contents are all the lines currently visible.

Solution 1: Let $\Lambda = \{L_1, \ldots, L_n\}$ be any collection of nonvertical lines. Assume without loss of generality that n is a power of 2. (If this is not the case and $2^p < n < 2^{p+1}$, then we may enlarge the set of lines by including $2^{p+1} - n$ lines parallel to L_1, each having a lower y-intercept than L_1. This transformation of the input doesn't change the set of visible lines, but increases the number of lines so that it is a power of 2.) Using divide-and-conquer, we will design an algorithm VISIBLE(Λ) that outputs a sequence of lines $L_{i(1)}, L_{i(2)}, \ldots, L_{i(k)}$ and intervals $I_1, I_2, \ldots, I_k$,[1] such that:

- $\{L_{i(1)}, L_{i(2)}, \ldots, L_{i(k)}\}$ is the set of visible lines in Λ.
- I_j is the set of x-coordinates at which $L_{i(j)}$ is uppermost.
- For $1 \leq j < k$, interval I_j lies to the left of interval I_{j+1}.

If $n = 1$ then VISIBLE(Λ) outputs the unique line in Λ, along with the interval $(-\infty, +\infty)$. Otherwise, the algorithm partitions Λ into two sets Λ_1, Λ_2 each having $n/2$ lines, and it runs VISIBLE(Λ_1) and VISIBLE(Λ_2) recursively. The output of VISIBLE(Λ_1) is a sequence of lines $L_{a(1)}, L_{a(2)}, \ldots, L_{a(p)}$ and a sequence of intervals $I'_1, I'_2, \ldots, I'_p$. The output of VISIBLE(Λ_2) is a sequence of lines $L_{b(1)}, L_{b(2)}, \ldots, L_{b(q)}$ and a sequence of intervals $I''_1, I''_2, \ldots, I''_q$. We then combine the lines using a procedure COMBINELINES which is similar to mergesort. The pseudocode for COMBINELINES is given below, in Algorithm 2. In the pseudocode, INTERSECT(L, L') denotes a procedure which takes two lines and outputs the coordinates (x, y) of their intersection point, or $(+\infty, +\infty)$ if they are parallel or if one of L, L' equals NULL. Computing INTERSECT(L, L') takes constant time.

Running time. The running time of COMBINELINES is $O(p + q)$ because every step takes constant time, and the number of iterations of the "while" loop is bounded by $p + q$ because every iteration results in one of the two linked-list pointers moving closer to the end of its list. Hence the running time of VISIBLE, $f(n)$, satisfies the recurrence

$$f(n) = 2f(n/2) + O(n)$$

whose solution is $f(n) = O(n \log n)$.

Correctness. The correctness of VISIBLE is a direct consequence of the correctness of COMBINELINES. To prove the correctness of COMBINELINES, the key fact that needs to be established is the following: if one considers a loop iteration that begins with the variable x being equal to some value x_0 and ends with x equal to some value x_1, then the algorithm's state at the start of that loop iteration satisfies the following for every $z \in [x_0, x_1]$:

- The set $\{\texttt{Vis}, \texttt{Invis}\}$ contains the line in $\{L_1, \ldots, L_p\}$ which is uppermost at z and the line in $\{L'_1, \ldots, L'_q\}$ which is uppermost at z.
- `Vis` lies above `Invis` at z.

This fact is established by induction on the number of loop iterations. The details of the inductive proof are a straightforward case analysis and are omitted here.

Solution 2: Compute the slope of each line, sort them in order of increasing slope, and retain only one line from each group of parallel lines, namely the one with maximum y-intercept. All

[1] An interval may have one or both endpoints in the set $\{\pm\infty\}$.

of these steps can be done in $O(n \log n)$ time. From now on, we may assume that $L_1, L_2, \ldots, L_n$ are arranged in order of increasing slope, and that no two of them are parallel.

The algorithm processes the lines in the order $L_1, L_2, \ldots, L_n$. Each time it processes a new line L_i, it intersects this line with the two topmost lines in the stack. (We assume that the stack implements a constant-time method `secondFromTop` that returns the element just below the top of the stack, assuming the stack depth is at least 2.) If the x-coordinate of the intersection point with the topmost line is to the right of the x-coordinate of the intersection point with the second line, then it pops the topmost line off the stack and recurses on the remaining stack contents. Otherwise it pushes L_i onto the stack. The pseudocode for this algorithm (excluding the preprocessing phase of sorting by slope and removing duplicate slopes) is given below in Algorithm 1.

Running time. The running time of ProcessLines is $O(n)$. To see this, note that each iteration of the "for" loop results in a `push` operation being performed on the stack, and each iteration of the "repeat" loop results in a `pop` operation. Each line is pushed onto the stack only once, and is popped at most once. Thus the total number of iterations of *both* loops, over the entire execution of the algorithm, is bounded by $O(n)$. Since the algorithm only goes a constant amount of work per loop iteration, the $O(n)$ running time bound follows. The time to compute all the visible lines is therefore $O(n \log n)$ because it takes $O(n \log n)$ time to sort the lines, and $O(n)$ time to process the sorted list using ProcessLines.

Correctness. The correctness of the algorithm is an immediate consequence of the following two lemmas.

Lemma 1. *If L_a, L_b, L_c are three lines arranged in order of increasing slope, and x_{ij} denotes the x-coordinate of the intersection point of L_i and L_j ($i, j \in \{a, b, c\}$), then x_{ac} lies between x_{ab} and x_{bc}.*

Proof. For $i = a, b, c$, suppose that L_i is the graph of $y = m_i x - r_i$. Then x_{ij} satisfies

$$m_i x_{ij} - r_i = m_j x_{ij} - r_j,$$

which implies

$$x_{ij} = \frac{r_j - r_i}{m_j - m_i}.$$

From this formula, we deduce that

$$x_{ac} = \left(\frac{m_b - m_a}{m_c - m_a}\right) x_{ab} + \left(\frac{m_c - m_b}{m_c - m_a}\right) x_{bc}. \tag{1}$$

Our hypothesis that $m_a < m_b < m_c$ ensures that the coefficients $\frac{m_b - m_a}{m_c - m_a}$ and $\frac{m_c - m_b}{m_c - m_a}$ are non-negative. Their sum is 1, so equation (1) expresses x_{ac} as a weighted average of x_{ab} and x_{bc}. Hence x_{ac} lies between x_{ab} and x_{bc}. □

Lemma 2. *At the end of processing line L_i, the stack in algorithm* ProcessLines *contains exactly the set of lines which would be visible if the input consisted only of the lines $L_1, \ldots, L_i$. These lines are arranged in order of increasing slope, with the greatest slope at the top of the stack. If (x, y) is the intersection point of the top two lines on the stack, then for all $x_0 \geq x$ the top line on the stack is uppermost at x_0 among lines $\{L_1, \ldots, L_i\}$.*

Proof. The proof is by induction on i. The base case $i = 1$ is trivial. To prove the induction step, let us say that a line L is *covered* by a set of lines Λ if it is the case that for every x, there is a line $L' \in \Lambda$ which lies above L at x. In other words, L is covered by Λ if L is not visible in the set $\{L\} \cup \Lambda$. Note that the coverage relation has the following important transitivity property: if L is covered by Λ, and every line in Λ is covered by Λ', then L is covered by Λ'.

We first claim that every line in $\{L_1, \ldots, L_i\}$ is covered by the set of lines in the stack at the end of processing L_i. We will establish this by proving that if a line L is popped from the stack during loop iteration i, then L is covered by L_i and the line L' that is second from the top of the stack at the time L is popped. (The claim in the first sentence of this paragraph then follows easily, using the transitivity property of coverage.) To prove this, let (x_1, y_1) and (x_2, y_2) be the intersection points of L_i with L and L', respectively, and let (x_3, y_3) be the intersection point of L with L'. The condition for L to be popped from the stack is that $x_1 \leq x_2$. Lemma 1 implies that x_2 lies between x_1 and x_3, so we may conclude that $x_1 \leq x_2 \leq x_3$. At every $x \leq x_3$, L' lies above L, whereas at every $x \geq x_1$, L_i lies above L. Thus $\{L', L_i\}$ covers L.

It remains to show that every line in the stack after processing L_i is visible in the set $\{L_1, \ldots, L_i\}$, and that the final two sentences in the statement of the lemma hold. The assertion that the lines on the stack are ordered by increasing slope after processing L_i follows trivially from the induction hypothesis that they are so ordered *before* processing L_i: the slope of L_i exceeds the slope of all lines underneath it, and the algorithm never changes the ordering of lines once they are sitting on the stack. Let L, L' denote the top two lines on the stack just before L_i is pushed onto the stack (with L sitting above L' on the stack), and let x_1, x_2, x_3 denote the x-coordinates of the intersection points of the pairs L_i and L, L_i and L', L and L', respectively. As before, we can use Lemma 1 to conclude that x_2 is between x_1 and x_3. Since the algorithm pushed L_i onto the stack, we know that $x_1 > x_2$, hence $x_2 \geq x_3$. For every $x \geq x_1$, L_i lies above L at x. By the induction hypothesis, L is uppermost among $\{L_1, \ldots, L_{i-1}\}$ at all $x \geq x_3$, so we see now that L_i is uppermost among $\{L_1, \ldots, L_i\}$ at all $x \geq x_1$. For all $x \leq x_1$, the line which is uppermost in $\{L_1, \ldots, L_{i-1}\}$ remains uppermost in $\{L_1, \ldots, L_i\}$ because L lies above L_i at these x-coordinates. (In particular, L itself remains uppermost for all x belonging to the interval $[x_3, x_1]$, which is nonempty because $x_3 \leq x_2 < x_1$.) Hence every line on the stack beneath L_i remains visible when L_i is pushed onto the stack. □

(3) We will design an algorithm for a slightly more general problem, which includes the given problem as a special case. Algorithm LocMinSearch(G, v) will take as input a grid G with n rows and m columns such that $|n - m| \leq 1$, and a node v in G, and it will output a node w such that w is a local minimum and the labels x_v, x_w satisfy $x_w \leq x_v$. The problem given in the textbook can be solved by calling LocMinSearch(G, v) where G is an $n \times n$ grid and v is an arbitrary node of G, e.g. the upper left corner.

Let $r = \lceil n/2 \rceil, c = \lceil m/2 \rceil$, and let G_1, G_2, G_3, G_4 be the four quadrants of G. (Each of these quadrants is specified by choosing a pair of rows $r_1 < r_2$ from $\{1, r, n\}$ and a pair of columns $c_1 < c_2$ from $\{1, c, m\}$. The quadrant consists of all nodes in rows r_1 through r_2 and columns c_1 through c_2. Note that the nodes in row r and those in column c belong to two or more quadrants.) In each of these quadrants, the number of rows differs from the number of columns by at most 1, so each G_i constitutes a valid grid on which we can recursively run LocMinSearch.

If $n \leq 3$, the algorithm exhaustively searches all nodes and outputs the global minimum. Otherwise, we use a divide-and-conquer approach as follows. First, the algorithm probes every node belonging to the boundary of any of the quadrants G_1, G_2, G_3, G_4, along with all neighbors of

every such node. Let w be the node whose label x_w is the minimum among the labels of all nodes probed so far. There are now several cases.

Case 1: If $x_w > x_v$, then v is contained in the interior of one of the quadrants G_i. We recursively call LocMinSearch(G_i, v) and output the result.

Case 2: If $x_w \leq x_v$ and w is on the boundary of a quadrant G_i, then we have already probed all neighbors of w in G, and (by the definition of w) their labels are all greater than x_w. Thus w is a local minimum satisfying $x_w \leq x_v$. The algorithm outputs w.

Case 3: If $x_w \leq x_v$ and w is in the interior of a quadrant G_i, then we call LocMinSearch(G_i, w) recursively and output the result.

Running time. The algorithm performs $O(n)$ probes, plus at most one recursive call of the same algorithm on a grid whose longest side has length at most $(n+3)/2$. Thus, the worst-case number of probes $f(n)$ satisfies

$$f(n) \leq f((n+3)/2) + O(n),$$

which implies $f(n) = O(n)$.

Correctness. We prove correctness by induction on n. In the base cases $n = 1, 2, 3$ we output the global minimum w, which is also a local minimum and satisfies $x_w \leq x_v$. Otherwise, the algorithm enters Case 1, 2, or 3 above. The correctness of Case 2 is clear. In Cases 1 and 3, we run LocMinSearch(G_i, u), where $u = v$ in Case 1 and $u = w$ in Case 3. In both cases, G_i is a quadrant of G and u is a node in the interior of G_i whose label x_u is less than the labels of all nodes on the boundary of G_i. By the induction hypothesis, LocMinSearch(G_i, u) outputs a node t which is a local minimum in G_i and which satisfies $x_t \leq x_u$. All labels on the boundary of G_i are greater than x_u, so t is in the interior of G_i, which means that every neighbor of t in G is also a neighbor of t in G_i. The fact that t is a local minimum in G_i thus implies that it is a local minimum in G, which completes the proof of correctness.

Algorithm 1 ProcessLines$(L_1, \ldots, L_n)$

Require: $L_1, \ldots, L_n$ are ordered by strictly increasing slope.

```
S = ∅
for i = 1, 2, ..., n do
  if S.depth ≥ 2 then
    (x1, y1) = Intersect(Li, S.top)
    repeat
      (x2, y2) = Intersect(Li, S.secondFromTop)
      if x1 ≤ x2 then
        S.pop
        x1 ← x2
      end if
    until x1 < x2 or S.depth < 2
  end if
  S.push(Li)
end for
```

Algorithm 2 CombineLines($L_1, \ldots, L_p; L'_1, \ldots, L'_q$)

Require: $L_1, \ldots, L_p$ are visible, in left-to-right order, when viewed from $y = +\infty$.
Require: $L'_1, \ldots, L'_q$ are visible, in left-to-right order, when viewed from $y = +\infty$.
Arrange $L_1, \ldots, L_p$ into a linked list $\mathscr{L}$.
Arrange $L'_1, \ldots, L'_q$ into a linked list $\mathscr{L}'$.
Initialize an empty linked list $\mathscr{O}$.
$L = \mathscr{L}$.`head`.
$L' = \mathscr{L}'$.`head`.
$x = -\infty$
// Let `Vis`, `Invis` be the lines which are visible, invisible at the current x-coordinate.
if Slope(L) < Slope(L') or L and L' are parallel and L has the higher y-intercept **then**
    `Vis` $= L$; `Invis` $= L'$
else
    `Vis` $= L'$; `Invis` $= L$
end if
// Main loop
while $x < \infty$ **do**
    (x_1, y_1) = Intersect(`Vis`, `Invis`)
    (x_2, y_2) = Intersect(`Vis`, `Vis.next`)
    (x_3, y_3) = Intersect(`Invis`, `Invis.next`)
    $x = \min\{x_1, x_2, x_3\}$
    if $x_1 = x$ **then**
        $\mathscr{O}$.`insertAtEnd`(`Vis`)
        `Tmp` $=$ `Vis`
        `Vis` $=$ `Invis`
        `Invis` $=$ `Tmp.next`
    else if $x_2 = x$ **then**
        $\mathscr{O}$.`insertAtEnd`(`Vis`)
        `Vis` $=$ `Vis.next`
    else
        `Invis` $=$ `Invis.next`
    end if
end while
return $\mathscr{O}$