CS 577: Introduction to Algorithms 9/21/2015

Homework 2 Solutions (Review Problems)

Instructor: Dieter van Melkebeek TA: Bryce Sandlund

Problem 1

In this problem, we are looking for an algorithm with running time $O(\log n)$. Regardless of what we are trying to do, this small running time does not allow very many options. This is good; it helps us narrow down the possible things that we might try. Running times with a factor of $O(\log n)$ should make you think of divide and conquer, and one of the few algorithms with running time $O(\log n)$ is BinarySearch.

However, we cannot use BinarySearch to directly solve this problem. In addition to the sorted array A and our target value x, another input to BinarySearch is the length of the array. Or more generally for the recursive version of BinarySearch, the additional inputs are indices i and j such that such that $0 \leq i \leq j \leq n-1$ and $A[i] \leq x \leq A[j]$ when x is in A. Of course we can pick $i = 0$, and we want to pick $j = n - 1$, but we don't know n.

So, can we find n in $O(\log n)$ time? If we can find k such that $A[k] = \infty$, then we know that $n < k$. Furthermore, we don't have to find n exactly. The Big-Oh notation $O(\cdot)$ hides constant factors and lower order terms, so the runtime of RecursiveBinarySearch is still $O(\log n)$ when $j = cn$ for any constant c.

Let's use the key idea from BinarySearch to find an index k such that $A[k] = \infty$ and k/n is small. BinarySearch quickly hones in on the target value by throwing away half the search space in each iteration. Our problem is that no matter what we start with as our initial guess as a bound on our search space, we might be too small. Since we have the opposite problem as BinarySearch, let's solve it by taking the opposite approach as BinarySearch. In each iteration, if we have yet to find an appropriate index k as described above, then we should *double* our search space in the next iteration. By dong this, in only $O(\log n)$ iterations, we can find an index k such that $A[k] = \infty$ and $n < k \leq 2n$.

Algorithm 1

Input: an infinite array A such that the subarray $A[0..n-1]$ contains integers sorted in non-decreasing order and the infinite subarray $A[n..]$ contains ∞, an integer x

Output: -1 if x is not in A, otherwise an index k such that $A[k] = x$

```
1: procedure InfiniteBinarySearch(A, x)
2:     m ← 1
3:     i ← 0
4:     while A[m] < ∞ do
5:         m ← 2m
6:         i ← i + 1
7:     return BinarySearch(m, A, x)
```

Formally, we run the procedure given in Algorithm 1. We prove correctness via partial correctness and runtime analysis (which implies termination).

Partial Correctness

An adequate invariant for the loop is that $m = 2^i$. Let m_j and i_j be the value of m and i respectively just before executing line 4 for the jth time. We prove our loop invariant by induction on j. The base case is $j = 1$. Then $m_j = m_0 = 1$ and $i_j = i_0 = 0$. Thus $m_j = 1 = 2^0 = 2^{i_j}$, so our invariant holds. For the inductive step, assume that $j > 1$ and $m_{j-1} = 2^{i_{j-1}}$. We need to show that $m_j = 2^{i_j}$. This is indeed true since

$$\begin{aligned} m_j &= 2m_{j-1} && \text{(line 5)} \\ &= 2 \cdot 2^{i_{j-1}} && \text{(inductive hypothesis)} \\ &= 2^{i_{j-1}+1} \\ &= 2^{i_j}. && \text{(line 6)} \end{aligned}$$

We use our loop invariant to prove partial correctness. Since m is always a power of 2 by our loop invariant, it is a valid index of A. Suppose we have reached line 7. Then the condition $A[m] < \infty$ is false, thus $A[m] \geq \infty$. However, there is nothing bigger than ∞, so actually $A[m] = \infty$. By assumption, n is the smallest index such that $A[n] = \infty$, so we have $n \leq m$.

The triple (m, A, x) is a valid input to BinarySearch, so in return, we either get -1 if x is not in $A[0..m-1]$ or k such that $A[k] = x$ otherwise. If x is not in the subarray $A[0..m-1]$, then x is not in x is not anywhere in A since $A[j] = \infty > x$ for all $j \geq m$ by assumption. Therefore, the value returned on line 7 from the call to BinarySearch is correct.

Runtime Analysis

Suppose we have reached line 7. Let m and i be the values of those variables at this point. We give a case analysis on whether or not the condition in line 4 was ever found to be true. If the condition was never true, then $m = 1$, and the runtime of BinarySearch is a constant, so the runtime of the entire procedure is a constant.

Otherwise, the condition was true at least once. Since the condition is currently false, it must have been true in the previous iteration, which means $A[m/2] < \infty$. Since $A[n] = \infty$, we have $m/2 < n$, or $m < 2n$.

Clearly the loop was executed i times. Recall from our loop invariant that we have $m = 2^i$. Thus, $i = O(\log m) = O(\log n)$. The runtime of the call to BinarySearch is $O(\log m) = O(\log 2n) = O(\log n)$. Therefore, the total runtime is also $O(\log n)$.

Problem 2

Approach Here is a natural divide-and-conquer approach:

1. Split up the lines into two groups of half the size: the left subproblem consisting of the lines $L_1, \cdots, L_m$, and the right subproblem consisting of the lines $L_{m+1}, \cdots, L_n$, where $m \doteq \lceil n/2 \rceil$.
2. Recursively solve the left subproblem and the right subproblem. Call those solutions $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$, respectively
3. Compute out of $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ the solution, say $\mathcal{L}$, for the full set of lines.

The key question is how to do step 3 efficiently. In particular, we'd need to execute it in $O(n)$ time in order to guarantee that our divide-and-conquer approach runs in time $O(n \log n)$.

Note that the set of x-coordinates for which a particular line L in $\mathcal{L}_{left}$ is uppermost among the lines in the left subproblem, forms an interval of the x-axis. All together, these intervals cover the entire x-axis except for the transition points where one interval ends and the next one starts, which are the intersection points of two lines in $\mathcal{L}_{left}$. The same holds for $\mathcal{L}_{right}$ and the right subproblem. For the full set of lines at a given x-coordinate, the only candidate uppermost lines we need to consider are the line that is uppermost at x in $\mathcal{L}_{left}$, and the line that is uppermost at x in $\mathcal{L}_{right}$; the one that is uppermost among the two of them is also uppermost among all of the lines. If a certain line, say L from $\mathcal{L}_{left}$ is uppermost at x among all of the lines, and we move further along the x-axis, L remains uppermost until either another line from $\mathcal{L}_{left}$ becomes uppermost, or we hit the intersection point with the line in $\mathcal{L}_{right}$ that is uppermost at x among the lines in the right subproblem. Based on this observation, if we had the lines in $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ sorted based on the interval where they are uppermost, we can execute step 3 in $O(n)$ time. Note that a line L comes before a line L' in this sorted order iff the slope of L is less than the slope of L'.

Sorting the lines in $\mathcal{L}_{left}$ and $\mathcal{L}_{right}$ at each level of recursion, would increase the amount of local work from $O(n)$ to $O(n \log n)$, resulting in an overall running time of $O(n(\log n)^2)$. However, similar to the solution in class for the problem of finding a closest pair of points, we can get around the need to sort explicitly at each level of recursion. There are two natural ways:

- Refine the output specification of our procedure by guaranteeing that the lines are output in sorted order.
- Restrict the input specification by imposing the additional precondition that the input lines are given in sorted order.

We will follow the second route. Note that in this approach, for small values of x, the uppermost line of the left subproblem is uppermost overall, for large values of x, the uppermost line of right subproblem is uppermost overall, and there is exactly one x-value, say x^*, where we transition from the left subproblem delivering the overall uppermost line to the right subproblem delivering the overall uppermost line. This follows from the fact that the slopes of all lines in the left subproblem are no more than the slopes of all the lines in the right subproblem.

In order to apply our procedure with the more restrictive input specification, we just sort the given set of lines from smallest to largest slope, spending $O(n \log n)$ time once.

Algorithm We first sort the lines in order of increasing slope, which can be done in $O(n \log n)$ using merge sort. Then for each set of parallel lines (i.e. those with the same slope), we only keep the one with the largest y-intercept. This can be done in $O(n)$ time. We then use a divide-and-conquer approach.

Our base case is $n = 1$, in which we simply return the given line. Otherwise, let $m \doteq \lceil n/2 \rceil$. We first recursively compute the sequence of visible lines among $L_1, \ldots, L_m$ to get lines $\mathcal{L}_{left} = \{L_{i_1}, \ldots, L_{i_p}\}$, which are still in order of increasing slope. We also compute within this recursive call the sequence of intersection points $a_1, \ldots, a_{p-1}$, where $a_k = L_{i_k} \cap L_{i_{k+1}}$. Notice that $a_1, \ldots, a_{p-1}$ have increasing x-coordinates since if two lines are both visible, the interval in which the line of smaller slope is uppermost is to the left of the interval in which the line of larger slope is uppermost. Similarly, we recursively compute the sequence of visible lines among $L_{m+1}, \ldots, L_n$ to get lines $\mathcal{L}_{right} = \{L_{j_1}, \ldots, L_{j_q}\}$ together with the sequence of points $b_1, \ldots, b_{q-1}$, where $b_k = L_{j_k} \cap L_{j_{k+1}}$.

To complete the algorithm, we show how to determine the visible lines in $\mathcal{L} = \mathcal{L}_{left} \cup \mathcal{L}_{right}$, together with their intersection points, in $O(n)$ time. Since $p + q \leq n$, it suffices to run in time $O(p + q)$. We know that L_{i_1} is visible because it has the minimum slope among all lines in $\mathcal{L}$.

Similarly, we know that L_{j_q} is visible because it has the maximum slope among all lines in $\mathcal{L}$.

We merge the sorted sequences $a_1, \ldots, a_{p-1}$ and $b_1, \ldots, b_{q-1}$ into a single sequence of points $c_1, c_2, \ldots, c_{p+q-2}$ ordered by increasing x-coordinate. This can be done in $O(n)$ time using the MERGE procedure discussed in lecture (a subroutine of merge sort). For each k satisfying $1 \leq k \leq p+q-2$, we consider the two lines $L_{i_s} \in \mathcal{L}_{left}$ and $L_{j_t} \in \mathcal{L}_{right}$ that are uppermost at the x-coordinate of c_k. Let $(x^*, y^*) = L_{i_s} \cap L_{j_t}$ be their intersection point. Let ℓ be the smallest index for which the uppermost line in $\mathcal{L}_{right}$ lies above the uppermost line in $\mathcal{L}_{left}$ at the x-coordinate of c_ℓ. Then x^* is between the x-coordinates of $c_{\ell-1}$ and c_ℓ. This means that L_{i_s} is uppermost in $\mathcal{L}$ immediately to the left of x^* and that L_{j_t} is uppermost in $\mathcal{L}$. immediately to the right of x^*. Therefore in $\mathcal{L}$, the sequence of visible lines is $L_{i_1}, \ldots, L_{i_s}, L_{j_t}, \ldots, L_{j_q}$ (in order of increasing slope) and the sequence of intersection points is $a_{i_1}, \ldots, a_{s_{s-1}}, (x^*, y^*), b_{j_t}, \ldots, b_{q-1}$. This completes the algorithm.

Analysis Correctness follows from the discussion. The running time of $O(n \log n)$ follows from having a binary recursion tree in which each child is only half the size of its parent, and spending a linear amount of time locally at every node.