

15-750-S06:

Midterm 1

February 27, 2006

Name:

Email:

Instructions

- Fill in the box above with your name, and your email address. **Do it, now!**
- This exam is open book: That is, you may use two books(Kozen and CLRS), any handouts from class, and your notes.
- Clearly mark your answers in the allocated space. If need be, use the back of a page for scratch space. If you have made a mess, cross out the invalid parts of your solution, and circle the ones that should be graded.
- Scan the test first to make sure that none of the 6 pages are missing. The problems are of varying difficulty, you might wish to pick off the easy ones first.
- You have 80 minutes. Good luck.

1	30	
2	30	
3	30	
Σ	90	

Problem 1: Silver Medal (30 pts)

Consider the following randomized algorithm for computing the first and second largest element for an unordered set of elements.

Input: a_1, \dots, a_n for $n \geq 2$ distinct elements

SILVERMEDAL(a_1, \dots, a_n)

```

1  randomly reorder the elements  $a_1, \dots, a_n$ 
2  set  $gold = -\infty$  and  $silver = -\infty$ 
3  for  $i \leftarrow 1$  to  $n$ 
4  do if  $a_i > silver$ 
5      then set  $silver = a_i$ 
6      compare and swap if necessary  $gold$  and  $silver$ 
```

The goal of this problem is to show that expected number of comparisons of the algorithm is $n + O(\log n)$.

- As a hints and also a subproblem. Let $S(i)$ be the probability that a_i is a silver medal contender. That is, $S(i)$ is the probability that a_i is assigned to *silver* in step 5 at some point in the algorithm.
- Show that expected number of comparisons of the algorithm is $n + O(\log n)$.
- Write the expected number of comparisons in the form $n + c \ln n + \Theta(1)$ for some fixed constant c .

SOLUTION: By observation, every element a_1, \dots, a_n is compared to *silver* exactly once, and every element a_i that becomes a silver medal contender is compared to *gold* exactly once. Thus, the number of comparisons the algorithm makes is $n + |C|$, where C is the set of nodes that become a silver medal contender.

As per the hint, we figure out the value of $S(i)$, the probability that a_i becomes a silver medal contender and observe that $E[|C|] = \sum_{i=1}^n S(i)$ by linearity of expectation. To figure out the value of $S(i)$, let $r(i)$ denote the rank of a_i if a_1, \dots, a_n are rearranged into sorted order. Notice that a_i is a member of C if and only if $|\{a_j \mid j < i \wedge r(j) < r(i)\}| \leq 1$. This occurs with probability 1 for the largest element, and with probability $2/r(i)$ for all other elements a_i for which $r(i) > 1$.

Thus, $E[|C|] = -1 + \sum_{i=1}^n 2/r(i) = -1 + \sum_{i=1}^n 2/i = 2 \ln n + \Theta(1)$, so the expected number of comparisons the algorithm makes is $n + 2 \ln n + \Theta(1)$.

Problem 2: Longest Monotone Subsequence (30 pts)

Let a_1, \dots, a_n be a sequence of values. We say that $a_{\sigma(1)}, \dots, a_{\sigma(k)}$ is a **nondecreasing monotone subsequence** if $a_{\sigma(i)} \leq a_{\sigma(i+1)}$ for $1 \leq i < k$.

The goal of this problem is to find the length of the longest nondecreasing monotone subsequence (LNMS). We shall solve this problem using dynamic programming. For simplicity we will assume that each a_i is an integer.

Assuming that each a_i is an integer in the set $\{1, \dots, n\}$, the goal is to construct a dynamic programming algorithm for the LNMS problem.

As always make sure you include in your dynamic programming solution the following:

- A short description of the subproblems you are solving
 - A recurrence for how the subproblems are solved
 - A proof of correctness
 - Timing analysis
- (a) Give an $O(n^2)$ algorithm for the LNMS problem for a sequence of length n .
- (b) Show how to improve your algorithm from above into an $O(n \log n)$ time algorithm.
- (c) For extra credit (only work on this problem after you have finished all the other problems): Suppose we have a data structure that supports the following operations for a set of n integers in the range 1 to n :
 INSERT, DELETE, PREDECESSOR, and SUCCESSOR.
 that works in $O(\log \log n)$ time per operation. Show how to use this data structure to give an $(n \log \log n)$ time algorithm for the LNMS problem.

SOLUTION:

- (a) As subproblems we let $f_i(j)$ be the length of the LNMS of a_1, \dots, a_i that includes the number j as the last member of the LNMS. If we define $f_0(j) = 0$ for all $j \in \{1, \dots, n\}$, we can define the following recurrence relation for $f_i(j)$ where $i, j \in \{1, \dots, n\}$:

$$f_i(j) = \begin{cases} \max_{1 \leq k \leq j} f_{i-1}(k) + 1 & \text{if } k = a_i \\ f_{i-1}(j) & \text{otherwise.} \end{cases}$$

Notice that the return value is $\max_{1 \leq j \leq n} f_n(j)$, so we do not need to keep f_{i-1} after computing f_i . Thus, if we represent f_{i-1} as an array of n elements, we can use the same array to represent f_i if we update the value of $f_{i-1}(a_i)$ to $f_i(a_i)$ as per the recurrence relation. The cost of this update is $O(n)$, so this technique yields an $O(n^2)$ algorithm.

- (b) Using the same recurrence, we can improve the running time to $O(n \log n)$ if we store a little extra information. In particular, suppose we store the elements $\{1, \dots, n\}$ in a balanced binary search tree T . Further, suppose we augment T so that each node j stores the maximum value of $f_i(k)$ such that k is in j 's subtree.

Then, when updating the value of $f_{i-1}(a_i)$ to $f_i(a_i)$, we can determine $\max_{1 \leq k \leq j} f_{i-1}(k)$ in $O(\log n)$ time using the maximum subtree values (using similar logic to how we computed the rank of elements in a BST by storing the sizes of the subtrees), and update the maximum subtree values of a_i and all of its ancestors.

- (c) Using the same recurrence, we can improve the running time to $O(n \log \log n)$ if we use our specialized data structure Q and are careful about how we solve the subproblems.

In addition to storing the array $f_i(j)$, we store in Q a *subset* of $\{1, \dots, n\}$. In particular, as we iteratively compute f_1, f_2, \dots, f_n we maintain the invariant at all times that j is a member of Q if and only if $f_i(k) < f_i(j)$ for all $k < j$. If we maintain this invariant, then $\operatorname{argmax}_{1 \leq k \leq j} f_{i-1}(k)$ equals the predecessor of j in Q immediately before updating to f_i from f_{i-1} (with some care chosen since j is not, strictly speaking, its own predecessor).

To maintain this invariant, in addition to performing the usual update from $f_{i-1}(a_i)$ to $f_i(a_i)$, we insert a_i into Q if it is not already in Q , and we find the successor of a_i and check whether it needs to be deleted from Q to maintain the invariant (convince yourself that this is the only element that may need to be deleted to satisfy the invariant).

All in all, each update consists of one predecessor query, up to one insert, one successor query, up to one delete, and a constant amount of work to update $f_{i-1}(a_i)$ to $f_i(a_i)$ in the array storing f_i , so our running time is $O(n \log \log n)$.

Problem 3: Visibility (30 pts)

You are given a collection of vertical line segments in the first quadrant of the $x y$ plane. Each line segment has one endpoint on the x -axis and the other endpoint has a positive y -coordinate. Imagine that from the top of each segment a horizontal bullet is shot to the left. The problem is to determine the index of the segment that is first hit by each of these bullet paths. If no segment is hit, return the index 0 (see Fig 1). The input is a sequence of top endpoints of each segment $p_i = (x_i, y_i)$ for $1 \leq i \leq n$, which are sorted in increasing order by x -coordinate. The output is the sequence of indices, one for each segment.

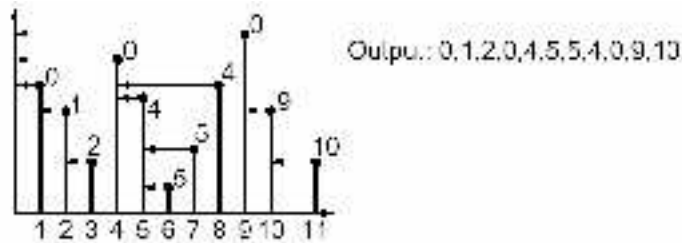


Figure 1: Visibility Problem

Present an $O(n)$ time algorithm to solve this problem using sweep line. Explain how your algorithm works and justify its runtime. Make sure you include answers to the following:

What are the events?

How are events handled?

What are the data structures used?

SOLUTION: We can use a left-to-right linesweep algorithm to solve this problem for which the events are the top end points p_i , sorted by increasing x -coordinate (as in the input to the algorithm). We handle each new event p_i by searching for the line segment that is visible to p_i , and deleting from future consideration the points that are in p_i 's "shadow" so that they can no longer be visible in any future event.

More specifically, our algorithm has the pseudocode below, where we assume for simplicity that each x_i and y_i is distinct and that each pair (x_i, y_i) is transformed into (i, y_i) in a linear time preprocessing step

```

VISIBILITY( $p_1, \dots, p_n$ )
1   $L \leftarrow$  an empty list
2   $S \leftarrow$  an empty stack
3  push  $(0, \infty)$  onto  $S$ 
4  for  $i \leftarrow 1$  to  $n$ 
5  do  $(i, y) \leftarrow$  the top element of  $S$ 
6     while  $y < y_i$ 
7     do pop  $S$ 
8      $(i, y) \leftarrow$  the top element of  $S$ 
9     append  $i$  to  $L$ 
10 return  $L$ 

```

Correctness of the algorithm follows because the elements of the stack are monotonically increasing and represent the sequence of line segments that do not completely fall within some other segment's shadow. Thus, when we successively pop S until we find an element p whose y value is not less than y_i , we know that p is visible to p_i .

This algorithm runs in $O(n)$ time because the outer **for** loop executes n times and the inner **while** loop executes once for every pop that occurs. There are at most n pops, so the algorithm runs in linear time.