Midterm Examination

Handed Out: Tuesday, March 3 1998
Due Date: Thursday, March 12 1998, 9:00am
This test is difficult. Please start early.

COLLABORATION POLICY

You can use any book, handout, class notes, etc., but you are not allowed to discuss the problems with other people or to get any other outside help. Feel free to contact the TAs or the instructor for questions regarding the statement of the problems; however they can not give you any direct help in solving the problems. Of course you may certainly ask them for clarifications regarding any topic that we have covered in the class.

PRINCETON UNIVERSITY HONOR CODE

Please sign below, and staple this cover sheet to your solutions.

"I have completed this midterm in accordance with the above policy."

SIGN YOUR NAME ABOVE THIS LINE

PRINT YOUR NAME ABOVE THIS LINE

(for grader's use only)

Problem	Points	Grade
1	25	
2	25	
3	25	
4	25	
Total	100	

Problem 1. (25 points)

Show how to maintain a dynamic set $\mathcal{Q}$ of numbers that supports the operation `Min-Gap`, which gives the magnitude of the difference of the two closest numbers in $\mathcal{Q}$. For example, if $\mathcal{Q} = \{1, 5, 9, 15, 18, 22\}$, then `Min-Gap`($\mathcal{Q}$) returns 3, since 15 and 18 are the two closest numbers in $\mathcal{Q}$. The operations required are `Insert`, `Delete`, and `Min-Gap`, where you may assume that a call to `Delete` includes some sort of handle to the location of the element being deleted.

All of your operations should run in $O(\lg n)$ time, in the worst case, and you should justify this fact.

Problem 2. (25 points)

Given a list of n rectangles which are axis-aligned (i.e., sides parallel to the x- and y-axis), you are to design an algorithm which detects whether any of the rectangles are overlapping each other. More specifically, your algorithm should either return `No`, in the case that no rectangles overlap, or else it should return `Yes`, and give an example of two rectangles which do overlap.

You should assume that each rectangle is represented by its minimum and maximum x- and y-coordinates, and that the list of rectangles is given to you in an arbitrary order. We will say that two rectangles overlap if any part of their interiors or borders touch each other.

A brute-force algorithm could determine this in $O(n^2)$ time by checking all possible pairs. Your algorithm should improve on this, running in $O(n \lg n)$ time, in the worst case. The description of your algorithm should be complete and unambiguous, however it should be described by high-level pseudocode, not low-level code. You should justify both the correctness of your algorithm as well as the bound on its running time.

(*Hint:* Consider moving a horizontal "sweep" line across the plane from top to bottom, and determine whether there are ever parts of two rectangles which intersect one another, on this line.)

Problem 3. Longest Monotone Sequence (25 points)

Given a sequence of numbers $X = \langle x_1, x_2, \ldots, x_n \rangle$, a *monotonically increasing subsequence* is a sequence $X' = \langle x_{a_1}, x_{a_2}, \ldots, x_{a_k} \rangle$ such that $1 \leq a_1 < a_2 < \ldots < a_k \leq n$ (i.e., the sequence is truly a subsequence), and that $x_{a_1} \leq x_{a_2} \leq \ldots \leq x_{a_k}$ (i.e., the sequence is monotonically increasing). Our goal is to find the *longest* monotonically increasing sequence. Actually, to make things easier, let's assume that we are only interested in knowing the *length* of the longest such sequence (although all of these techniques can be extended to build the actual sequence).

For example if $X = \langle 5, 1, 4, 2, 3, 8, 6, 7 \rangle$, then the longest monotonically increasing subsequence is $\langle 1, 2, 3, 6, 7 \rangle$, with length 5.

To design a dynamic programming algorithm we will consider the following subproblem for each $k \in \{1, \ldots, n\}$, namely, what is the longest monotonically increasing subsequence which ends with x_k. If we let $L(k)$ equal the *length* of the longest such sequence which ends with x_k, we claim the following recursive formula exists,

$$L(k) = 1 + \max_{\substack{j \in \{1,\ldots,k-1\} \\ \text{with } x_j \leq x_k}} L(j) \tag{1}$$

a. Prove Equation (1).
   (Make sure you prove both the "$\geq$" and the "$\leq$" implied by equality.)

b. Show that the *length* of the longest monotonically increasing sequence can be computed using dynamic programming based directly on Equation (1). Show that this can be done in $O(n^2)$ time.

Additional algorithmic techniques can often be used to speed up the running time of a dynamic program. This is the case with this problem, as we will see how to improve our algorithm so that it runs in $O(n \lg n)$ time. To do so, we make the following observation,

Lemma 1. *Let $S_k \subseteq \{1, \ldots, k-1\}$ be a subset containing both a and b. If $L(a) = L(b)$ and $x_a \leq x_b$, then*

$$\max_{\substack{j \in S_k \\ \text{with } x_j \leq x_k}} L(j) = \max_{\substack{j \in S_k - \{b\} \\ \text{with } x_j \leq x_k}} L(j)$$

c. Prove Lemma 1.

d. Use Lemma 1 to show that there exists a set $C_k \subseteq \{1, \ldots, k-1\}$, which we will call "candidates", such that

$$L(k) = 1 + \max_{\substack{j \in C_k \\ \text{with } x_j \leq x_k}} L(j) \tag{2}$$

with the additional condition that if $a \in C_k$ and $b \in C_k$ then $L(a) \neq L(b)$.

Since part d states that set C_k contains at most one candidate for each possible length, we can maintain an array $candidate[1..n]$, relative to set C_k, with entries

$$candidate[l] = \begin{cases} x_a & \text{if } a \in C_k \text{ with } L(a) = l \\ \infty & \text{if no such } a \text{ exists} \end{cases}$$

e. Prove that the array $candidate[1..n]$ is sorted in non-decreasing order, that is, show that $candidate[r] \leq candidate[s]$ if $r < s$ for $r, s \in \{1, \ldots, n\}$

f. Show how we can use the array $candidate[1..n]$, together with Equation (2), in order to compute $L(k)$ in $O(\lg n)$ time.

g. What does set C_1 look like? How should we initialize the array $candidate[1..n]$ to correspond to set C_1?

h. How does set C_k compare to set C_{k+1}? After computing $L(k)$, how quickly can we update the array $candidate[1..n]$ to correspond to set C_{k+1} rather than set C_k?

i. Explain why we are now able to calculate the length of the longest monotonically increasing subsequence in $O(n \lg n)$ time overall.

Problem 4. Binary Search Trees (25 points)

Assume that we have a list of n keys, $\langle a_1 \le a_2 \le \ldots \le a_n \rangle$, and our goal is to construct a static binary search tree, which will later be used for searches. In addition, assume that we know the probability, p_i, that a search will request the key a_i. If all searches are for objects from the original set, then we know,

$$\sum_{i=1}^{n} p_i = 1.$$

Our goal is to pick an optimal binary search tree which will minimize the *expected cost* of a search.

a. Assume that the cost of searching for a_i in binary search tree T is exactly $d_T(a_i) + 1$, where $d_T(a_i)$ is the depth of a_i in T. Give an expression for the expected cost of a search in tree T.

b. A possible algorithm, `Greedy`, for designing a tree, picks a node with the highest search probability, makes that node the root of our tree, and recursively builds the left and right subtrees.

   Give a *simple* example for which the `Greedy` algorithm produces a tree which is not optimal.

For the rest of this problem, we would like you to develop and analyze a bottom-up dynamic program which constructs an optimal binary search tree.

c. Formally define the set of subproblems which you will consider, and explain what information you plan to store for each such subproblem.

d. Give a recursive definition which explains how you will calculate, for each subproblem, all of the information you mention in part c. Please make sure your recursive definition includes any relevant base cases.

e. Prove the correctness of the recursive definition you gave in part d.

f. Give a bottom-up order which you can use to solve the subproblems, based on your recursive definition from part d.

g. Explain how the information you have gathered can be used to return the final tree T, as well as the value of an expected search in that tree.

h. Give a tight asymptotic bound for the overall running time of your algorithm. (explain your answer)