CSOR W4231 (sec. 001, 002, H02): Analysis of Algorithms Oct 15

Homework 4: Due on Oct 31 by 12:01am

Instructors: *Alex Andoni, Cliff Stein*

Instructions. Please follow the homework policy and submission instructions in the Course Information handout. A few highlights to remember:

- follow the collaboration and academic honesty policies;
- write your name, uni, and collaborators on the top page;
- submission is via GradeScope (you are encouraged to use LaTeX, using the provided template);
- if you don't know how to solve a particular part of a problem, just write "*empty*", for which you will get 20% of the points of that part (in contrast, note that non-sensical text may get 0%).

Note that, for each bullet item of a problem, you can use the previous bullets as "black box", even if you didn't manage to solve them. Similarly, you can use anything from the lectures as black-box.

1 Problem 1 (20pts)

Describe an efficient algorithm that, given a set $\{x_1, x_2, ..., x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct, and analyze the running time.

2 Problem 2 (30pts)

Consider the generalization of the activity selection problem, where each activity a_i has a given positive weight w_i, and we want to select a subset Q of non-overlapping activities with maximum total weight $w(Q) = \sum_{a_i \in Q} w_i$.

(a) Give counterexamples showing that the greedy algorithm with either of the following two selection criteria does not always produce optimal solutions: (i) Select an activity of largest weight. (ii) Select an activity with the earliest finishing time.

(b) Suppose that the activities are ordered in order of finishing time (earliest first). Let $W(i)$ denote the maximum weight of a non-overlapping subset of the first i activities. Show how to compute $W(i)$ from the set of $W(j)$ with $j < i$.

(c) Give an algorithm that computes an optimal solution, i.e., a subset of non-overlapping activities with maximum total weight. Be sure to argue that it is correct and analyze the running time.

3 Problem 3 (30pts)

A time-series is a sequence of integers $a_1, a_2, \ldots, a_n$, where a_i represents (some) signal at time i. Oftentimes, we need to be able to compare two time-series and declare whether they are similar or not (e.g., imagine these are the time-series of heart beats: of a patient, and of "normal heart beat"). One way to compare two such time series, called $a_1, a_2, \ldots a_n$ and $b_1, b_2, \ldots b_m$ is as follows. Define *a matching* to be a non-decreasing map $f : [n] \to [m]$, thought of as a map from indices of a to indices of b, such that $f(i) \leq f(j)$ whenever $i < j$. For a given matching f, the cost of the matching is $cost(f) = \sum_{i=1}^{n} |a_i - b_{f(i)}|$.

Finally, the distance between $a_1, a_2, \ldots a_n$ and $b_1, b_2, \ldots b_m$ is the minimum, over all non-decreasing matchings f, of $cost(f)$. For example, for sequences $4, 3, 8, 8, 1$ and $3, 4, 8, 2$, the distance is 0+1+0+0+1=2, obtained by the matching $f(1) = 2, f(2) = 2, f(3) = 3, f(4) = 3, f(5) = 4$ (among a few matchings f that get this optimal value).

Design an algorithm for computing the distance between two given sequences $a_1, a_2, \ldots a_n$ and $b_1, b_2, \ldots b_m$.

(a) Show a naïve algorithm runs in time $O(m^n \cdot n)$. You do not need to analyze correctness or runtime.

(b) Let $C(i, j)$ be the distance between $a_1, a_2, \ldots a_i$ and $b_1, b_2, \ldots b_j$. Deduce a recursive formula for $C(i, j)$: i.e., how to compute $C(i, j)$ from values $C(i', j')$ for $i' \leq i, j' \leq j$ (where at least one inequality is strict).

(c) Now, using part (b), design and analyze an algorithm, running in time $O(nm^2)$, for computing the distance between two given sequences $a_1, a_2, \ldots a_n$ and $b_1, b_2, \ldots b_m$. Alternatively, design an algorithm with $O(n^3)$ runtime when $m = n$.

Bonus: Show that you can implement the algorithm from part (c) in $O(nm)$ time only (or $O(n^2)$ time for the $m = n$ case).

4 Problem 4 (20pts)

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n + 1) \rceil$, and let the binary representation of n be $< n_{k-1}, n_{k-2}, \ldots, n_0 >$. You maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k - 1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \, 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

1. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

2. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

3. Describe how to implement DELETE. Analyze its worst-case and amortized running times, assuming that there can be DELETE, INSERT, and SEARCH operations.