Foundations of Algorithms, Spring 2022: Homework 4

Due: Wednesday, March 23, 11:59pm

Problem 1 (8 points)

Given is a sequence of distinct positive numbers. We want to find a subsequence with the maximum possible sum, with the restriction that we are not allowed to take three consecutive elements from the original sequence. For example, for input $1, 6, 5, 2, 7, 9, 3, 4$, the subsequence with the maximum possible sum is $6, 5, 7, 9, 4$ (we have two pairs of consecutive elements $6, 5$ and $7, 9$ but not three consecutive elements).

Now consider the following greedy strategies:

- Start with an empty subsequence. Go through the numbers in the sequence in decreasing order. For every number, if it does not cause three consecutive elements in the subsequence, include it in the subsequence (otherwise skip the number). For our example, we include $9, 7, 6, 5, 4$, then we skip $3, 2$, and 1.
- Go through the numbers in the sequence in increasing order, crossing them out until you get a sequence without three consecutive elements. For our example, we cross out $1, 2$, and 3.
- Split the input into groups of three elements (the last group may have one or two elements). Go through the groups from left to right. For each group find the maximum element and include it in the subsequence. After that step is completed, go through the remaining elements in decreasing order, including an element if it does not cause three consecutive elements in the subsequence (otherwise skipping the element). For our example, the groups are $1, 6, 5$ and $2, 7, 9$ and $3, 4$. We include $6, 9$ and 4 in the subsequence. Then we include 7 and 5, and skip $3, 2$, and 1.

None of these greedy strategies always produces the optimal answer. For each greedy strategy, provide a counterexample input, along with an optimum solution and its sum (for this input), and the solution produced by the greedy algorithm and its sum (for this input).

Problem 2 (15 points: 10 for implementation / 5 for writeup)

Give an $O(n)$ dynamic programming solution to Problem 1 (assume input sequence of length n). Remember that your written portion for dynamic programming problems such as this should describe your "heart of the algorithm," as well as a complexity analysis.

Problem 3 (12 points: 7 for implementation / 5 for writeup)

Consider the following problem related to ordering playing cards in your hand. You hold n cards in your hand (you have a big hand; n can be an arbitrarily large number). The n cards have distinct integer values in the range $1, 2, \ldots, n$. The cards have been shuffled, so you initially hold them in a random ordering.

Design an $O(n^2)$ algorithm that determines the minimum number of cards that need to be moved in order to get the whole set of n cards in sorted, ascending order. One move corresponds to extracting one card from its current location, and moving it to a different location. In the process, the remaining cards shift as necessary in your hand to make room (this shifting does not count as any movement).

For example, if you hold the cards $2, 3, 1, 5, 4$ in your hand, it will take 2 moves to arrange the cards in sorted, ascending order. The 1 card can be moved to the leftmost position. The 4 card can be moved left of the 5.

Problem 4 (20 points: 13 for implementation / 7 for writeup)

A sister and brother run a summer babysitting business. They have a lot of job offers, and are deciding which jobs to take to maximize their income. Each job comes with the following information: the day of the job, the starting time of the job, the ending time of the job, the number of children that need to be babysat, and the hourly rate for the job. Brother and sister can work separate jobs, except for one case: any job involving four or more children requires them to work together if they take that job. In order to make sure they have time to sleep, they must end each day by 11pm, and start no earlier than 6am the next day.

Assuming that sister and brother are presented with n total jobs to choose from, design an $O(n^2)$ algorithm that determines the most money that they can earn. As your argument of correctness, state all three parts of the heart of your algorithm.

Note: although $O(n^2)$, your algorithm should be designed to handle the provided large input cases without memory issues and without exceeding one minute execution time.

Problem 5 (18 points: 12 for implementation / 6 for writeup)

There is a variation of the longest common subsequence problem in which we are interested in the largest possible *sum* of a common subsequence, instead of the largest number of elements included.

For this problem, we will consider a twist on this largest possible sum problem in which we loosen the requirement that the elements chosen are exact matches.

Given are two arrays of length n of positive integers: $A = [a_1, a_2, ..., a_n]$ and $B = [b_1, b_2, ..., b_n]$. Design an $O(n^2)$ algorithm that determines the maximum sum possible of a subsequence of A such that there exists a subsequence of B that includes the same number of elements (not necessarily corresponding to the same indices), has the same sum, and for which the partial sums of the elements of A subsequence never differ by more than one from the corresponding partial sums of the elements of the B subsequence.

For example, given $A = [2, 6, 3, 4, 10]$ and $B = [1, 7, 8, 9, 5]$. We can find a largest sum of 18 that satisfies the required constraints, by choosing $\{2, 6, 10\}$ from A and choosing $\{1, 8, 9\}$ from B. The partial sums of the elements of each chosen subsequence are: 2 compared to 1, $2 + 6 = 8$ compared to $1 + 8 = 9$, and finally $2 + 6 + 10 = 18$ compared to $1 + 8 + 9 = 18$. In each case, the partial sum of the elements of the A subsequence differs by no more than one from the corresponding partial sum of elements of the B subsequence. And the total sum of the subsequences is an exact match.