Homework 6 – Due Thursday, October 13, 2016 on Canvas

Please refer to HW guidelines from HW1, course syllabus, and collaboration policy.

Exercises These should not be handed in, but the material they cover may appear on exams:

1. (**Dynamic programming**) Starred exercises are harder.

   (a) (**Segmented LS with exactly k pieces**) Consider the following variant of the Segmented Least-Squares Problem (KT Chapter 6.3): given n points $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$ with distinct x coordinates, we want to compute a piece-wise-linear approximation to the point set *with exactly k pieces* that minimizes the sum squared error (that is, rather than minimizing $E + C \cdot L$ as in class, we want to minimize E subject to $L = k$).

   Your algorithm should run in time $O(kn^2)$. You may assume that the numbers e_{ij} used in the algorithm in class have already been computed.

   (b) The algorithm we saw in class for the segmented least squares problem requires the quantity $e_{i,j}$ (the best squared error of any single-line for points i through j). Give a $O(n^2)$-time algorithm to simultaneously compute all the values $e_{i,j}$. (*Hint: See footnote 1 on page 266. The formula for $e_{i,j}$ is given in the lecture notes and on page 262 of the book.*)

   (c) Exercises in KT, Chapter 6; exercises in Erickson's notes on dynamic programming.

   (d) Design and analyze an algorithm that takes an array A of n numbers (some of which may be negative) and finds indices i, j for which the sum of the elements in $A[i..j]$ is as large as possible.

   (e) Design and analyze an algorithm that takes an array A of n numbers (some of which may be negative), and finds indices i, j for which the average of the elements in $A[i..j]$ is as large as possible.

   (f) Given an array A of numbers, find the longest increasing subsequence. A sequence of indices $i_1 < i_2 < \cdots < i_k$ is increasing if $A[i_1] \leq A[i_2 \leq \cdots \leq A[i_k]$. The straightforward solution has complexity $O(n^2)$; after you've understood it, try to find a solution that runs in time $O(n \log n)$.

   (g) You are given a collection of n line segments in the Euclidean plane $\mathbb{R}^2$, where the ith segment goes from the point $(s_i, 0)$ to $(t_i, 1)$ (note that the end points of the segments always lie on the two parallel lines defined by $y = 0$ and $y = 1$). You may assume, for simplicity, that the values s_i, $i = 1, ..., n$, are all distinct, as are the values t_i (so that intersections never occur at the endpoints of the line segments). Give an algorithm which finds the largest set of non-intersecting line segments in the input collection. Explain why it is correct and analyze its running time.

   (h) A subsequence $i_1 < i_2 < \cdots < i_k$ of an array of numbers is *oscillating* if $A[i_j] \leq A[i_{j+1}]$ when j is odd and $A[i_j] \geq A[i_{j+1}]$ when j is even. Give an algorithm which takes an array of numbers and finds the longest oscillating subsequence. (NB: This problem also admits a greedy solution.)

   (i) Give an algorithm that takes two strings and finds a shortest common supersequence (that is, a shortest string in which both inputs are subsequences).

(j) Give an algorithm that takes as input a DAG with edge weights as well as two vertices s, t and finds the length of the longest path from s to t (assuming that one exists).

(k) * **(Saving space in dynamic programming)** Design and analyze an algorithm which solves the knapsack problem in $O(n + W)$ space and $O(nW)$ time. [Hint: modify the linear-space algorithm for the least common subsequence.]

(l) * A $n \times n$ black-and-white bitmap image is stored as an array A in which each entry gives the color (either black or white) of the corresponding pixel. A *monochromatic block* is a submatrix of the form $A[i, .., k][j, ..., l]$. Give an algorithm which finds the largest monochromatic block. Explain why it is correct and analyze its running time.

(m) * (**Longest well-colored path**) Suppose you are given an undirected graph G, along with a coloring function $c : V \to \{1, 2, ..., k\}$ that assigns one of k possible colors to each vertex. We say a path in G is well-colored if every vertex along the path (including the start and end points) has a different color. Of course, well-colored paths can have length at most $k - 1$.

Give a $O(2^k \cdot \text{poly}(n))$-time algorithm that outputs the longest well-colored path in G. (Here, poly(n) can be any fixed polynomial in n that doesn't depend on k.)

2. (**DFT**)

(a) Compute the DFT of the vector $(0, 1, 2, 3)$.

(b) Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using the FFT scheme. Verify your answer by directly multiplying the polynomials using the straightforward algorithm that runs in $\Theta(n^2)$ time.

Problems to be handed in (Don't forget to prove correctness and analyze time/space requirements of your algorithm.)

1. (**Longest path in an ordered graph**) Chapter 6, problem 3.

2. **(Cheapest Festival Cover)** Recall the problem from the first midterm where you were asked to design a greedy algorithm for scheduling the smallest number of volunteers to cover a festival. In this problem, you are asked to design a dynamic programming algorithm for scheduling a cheapest subset of workers to cover a festival.

   You know the start s and the finish f time of the festival. Each worker i gave you her arrival time a_i, the time b_i she must leave, and the amount p_i she wants to get paid. Your goal is to select the cheapest set of workers who can cover the entire festival, so that at least one worker is always there. (If one worker is arriving at the same time as another is leaving, it is ok.) You may assume for simplicity that (i) all the endpoints a_i, b_i are distinct, (ii) at each point in time, there is at least one worker willing to work.

   The natural solution runs in time $\Theta(n^2)$ in the worst case. If you can, give a (correct, nicely explained) solution that runs in time $o(n^2)$. (Don't forget to output the workers you are scheduling, not only the cost of hiring them.)

3. (**Carpenter's ruler**) A *carpenter's ruler* consists of several links that are hinged together at their endpoints. We will model the links as n line segments of (0 width and) integer lengths $\ell_1, \ell_2, \ldots, \ell_n$ that may rotate freely about their joints.

When we fold a carpenter's ruler, each pair of consecutive links can form either 0° or a 180° angle at the joint between them. A *folding* of a ruler can be specified by a binary string of length n in which the i-th bit is 0 or 1, depending on whether the i-th segment is folded to the left or to the right of its fixed endpoint (view this as a sequential process). The *length* of the folding is the length of the smallest closed interval that contains all segments of the ruler.

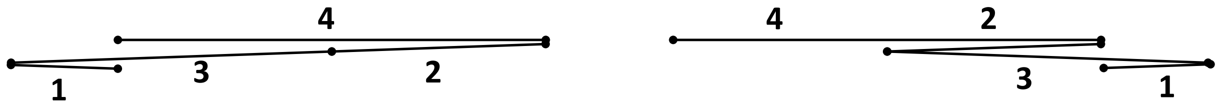

Figure 1: Two ways to fold a carpenter's ruler with link lengths 1,3,2,4 into length 5. The left folding is specified by string 0110; the right folding, by 1010.

Let L be the length of the longest link, that is, $L = \max\{\ell_i : 1 \leq i \leq n\}$.

(a) Prove that a carpenter's ruler can always be folded into length at most $2L$.

(b) Design an efficient dynamic programming algorithm that, given integers n and $\ell_1, \ell_2, \ldots, \ell_n$, computes the smallest folding length of a carpenter's ruler consisting of n segments of the specified lengths. Your algorithm should take $O(L^3 n)$ time. (If you can, give an $O(L^2 n)$ algorithm.) You may use the statement from part (a) even if you did not prove it.

(c) Give an efficient algorithm that produces a minimum-length folding (specified by a binary string of length n, as explained above).

- (**Do not hand in**) Does your algorithm take polynomial time?