CS 161 **Problem Set 6**
Summer 2017 **Due:** August 13, 2017, 11:59 p.m.

---

1. **Allocating Surfboard.** (4 points)

   A group of $n$ friends have respective heights $h_1 < h_2 < ... < h_n$ (where $h_i$ is the height of friend i). They decide to go surfing and need to rent surfboards. The surf shop has a rack with $m > n$ surfboards ordered by lengths $s_1 < s_2 < ... < s_m$. In small/clean waves, the ideal surfboard has the same length as your height. Help us figure out a good allocation of the boards.

   Formally, an allocation of surfboards is a function $f : \{1, ..., n\} \to \{1, ..., m\}$ that maps each surfer to a surfboard. More precisely, $f(2) = 3$ means that surfer 2 (with height $h_2$) receives surfboard 3 (with length $s_3$). An allocation $f$ is optimal if it minimizes the quantity $\sum_{k=1}^{n} |h_k - s_{f(k)}|$. That is, an allocation is optimal if it minimizes the sum of the discrepancies of height between the surfers and their surfboards.

   Let $A[n, m]$ denote this minimal difference.

   (a) (2 point) Let $A[i, j]$ denote the sum of discrepancies of an optimal allocation of the first $j$ surfboards to the first $i$ surfers ($j \geq i$). Prove that, if surfboard $j$ is used in an optimal allocation, then there is an optimal allocation in which it is allocated to surfer $i$.

   Note: There might be multiple optimal allocations. This part asks you to show that if the longest board is used, then it might as well go to the tallest surfer.

   [**We are expecting a formal proof of your answer**]

   (b) (1 point) Deduce a recurrence relation between $A[i, j], A[i, j-1]$ and $A[i-1, j-1]$.

   Hint: Consider two cases, according to whether surfboard $j$ is used or not.

   [**We are expecting a statement of the recurrence as well as a short explanation of it**]

   (c) (3 points) Design a dynamic programming algorithm that computes $A[n, m]$ and also outputs the optimal allocation.

   [**We are expecting a description of a procedure or pseudocode of an algorithm**]

   (d) (1 point) What is the runtime of your algorithm? Prove your answer.

   [**We are expecting an informal analysis of the runtime**]

2. **Pruning Trees** (6 points) Suppose you are given a tree with $n$ nodes, and each node has an associated weight. You would like to remove nodes from this tree such that the resulting tree has exactly $k$ nodes, and you would like to maximize the sum of the weights of the remaining nodes. Additionally, at the end of the pruning, you must still have a tree rooted at the original root; in other words, if you remove a node, then all of that node's children/descendants must also be removed. For parts (a) (b) and (c) below, you can assume the tree is a binary tree.

(a) (1 point) For any node $u$ of the original tree, and any integer $i \leq k$, let $A[u, i]$ denote the maximum weight of any subtree rooted at node $u$ having exactly $i$ nodes. Letting $r_u$ and $\ell_u$ denote the right and left children of node $u$, formally describe the optimal structure by giving a recurrence that expresses $A[u, i]$ in terms of the quantities $A[r_u, 1], A[r_u, 2], \ldots$ and $A[\ell_u, 1], A[\ell_u, 2], \ldots$. [You can assume the tree is a binary tree.]

[**We are just expecting an expression for** $A[u, i]$]

(b) (4 points) Leveraging the insights from the previous problem, define a dynamic program that will efficiently solve the problem. Please have your program return both the weight of the best tree, as well as a description of the optimal tree. Provide a brief justification for the correctness of the algorithm. [You can assume the tree is a binary tree.]

[**We are expecting a DP formulation and a correctness proof.**]

(c) (1 points) What is the runtime of your algorithm? [You can assume the tree is a binary tree.]

[**We are expecting the runtime and a brief description of how you arrived at your answer.**]

(d) (2 point bonus) Give an algorithm for this problem that works for trees with arbitrary branching factor (i.e. not necessarily binary trees), which has the same Big-Oh worst-case runtime as the above dynamic programming algorithm. [Hint: try to convert a non-binary tree into a binary tree, and modify your dynamic program from part (b) to correctly deal with this new binary tree...kindof : )

[**We are not expecting anything. It is a bonus question.**]

3. **Taking Stock** (8 points)

Suppose you are given reliable insider information about the prices for $k$ different stocks over the next $n$ days. That is, for each day $i \in [1, n]$ and each stock $j \in [1, k]$, you're given $p_{i,j}$, the price at which stock $j$ trades at on the $i$th day. You start with a budget of $P$ dollars, and on each day, you may purchase or sell as many shares of each type of stock as you want, as long as you can afford them. (Assume that all prices are integer-valued and that you can only purchase whole stocks.) You have an earning goal of $Q$ dollars. Here, we will design an algorithm to determine whether you can meet your goal, and if not, how much money you can earn.

(a) (3 points) Suppose we are only looking at prices over two days (i.e. $n = 2$). Design an $O(kP)$ dynamic programming algorithm that computes the amount of money you can make buying stocks on the first day and selling stocks on the second day. Prove the runtime and correctness of your algorithm. (*Hint: Let $M[l]$ be the amount of money you can make after investing l dollars. To start, write an expression for $M[l]$ in terms of $M[l']$ for $l' \leq l$.*)

(b) (5 points) Now, suppose you are given prices over $n$ days. Using your solution to part (a) as a guide, design an $O(nkQ)$ time algorithm that determines whether you can reach your goal, and if not, reports how much money you can make. Prove your algorithm's runtime and correctness. (*Hint: Without loss of generality, you can imagine at the start of every day, you sell every stock you own, and purchase stocks with your correct earnings.*)

4. **Toll Game.** (5 points **extra credit**, edited on 8/7)

   Consider the following game on a weighted directed graph $G = (V, E)$. You start at a given vertex $s$. At each step you traverse some edge $(x, y)$ paying $w(x, y)$ dollars. (The number $w(x, y)$ may be negative, but if you get stuck at a vertex with no outgoing edges,your have to pay an infinite fine.) You play this game for a very long time. Find an algorithm to compute the minimum average cost per move as the number of moves goes to infinity. The running time should be $O(nm)$, where $n$ is the number of vertices, and $m$ is the number of edges.

   **Hint:** The solution is based on so-called duality: instead of minimizing a function, one can look for a tight lower bound. More concretely, let $\lambda_*$ be the minimum average cost, $\lambda$ an arbitrary number, and let us ask this question: under what circumstances $\lambda \leq \lambda_*$? Here is an initial answer:

   $\lambda \leq \lambda_*$ if there is no reachable negative-weight cycle with respect to the weight system $w_\lambda(u, v) = w(u, v) - \lambda$

   (explain why). The main part of the solution is to cast this condition in such a form that would allow for the maximization of $\lambda$. To this end, you may use these functions:

   $$g_k(x) = min\{w(p) : p \text{ is a path from } s \text{ to } x \text{ with exactly } k \text{ edges}\},$$
   $$\tilde{g}_{\lambda,k}(x) = min\{w_\lambda(p) : p \text{ is a path from } s \text{ to } x \text{ with at most } k \text{ edges}\}.$$

   Here $w(p)$ denotes the path length, $w(x_0, ..., x_k) = \sum_{j=1}^{k} w(x_{j-1}, x_j)$, and $w_\lambda(p)$ is defined similarly. Check if $\tilde{g}_{\lambda,n}(x) = \tilde{g}_{\lambda,n-1}(x)$ for all $x$. But do that analytically, treating $\lambda$ as a variable. Computation is only possible at the very last step.

   [**We are expecting a description or pseudocode of your algorithm as well as a brief justification of its correctness and runtime.**]

5. **Currency Exchange, revisited** (4 points, added on 8/7)

   Recall the problem statement from Homework 4, Question 3a: Suppose the various economies of the world use a set of currencies $C_1, \ldots, C_n$—think of these as dollars, pounds, bitcoins, etc. Your bank allows you to trade each currency $C_i$ for any other currency $C_j$, and finds some way to charge you for this service (in a manner to be elaborated in the subparts below). We will devise algorithms to trade currencies to maximize the amount we end up with.

   (a) (2 points) Consider the more realistic setting where the bank does not charge flat fees, but instead uses exchange *rates*. In particular, for each ordered pair $(C_i, C_j)$, the bank lets you trade one unit of $C_i$ for $r_{ij} > 0$ units of $C_j$. Devise an efficient algorithm which, given starting currency $C_s$, target currency $C_t$, and a list of rates $r_{ij}$, computes a sequence of exchanges that results in the greatest amount of $C_t$.

Justify the correctness of your algorithm and its runtime.[Hint: How can you turn a product of terms into a sum? Take logarithms.]

(b) (2 points) Due to fluctuations in the markets, it is occasionally possible to find a sequence of exchanges that lets you start with currency A, change into currencies, B, C, D, etc., and then end up changing back to currency A in such a way that you end up with more money than you started with—that is, there are currencies $C_{i_1}, \ldots, C_{i_k}$ such that

$$r_{i_1 i_2} \times r_{i_2 i_3} \times \cdots \times r_{i_{k-1} i_k} \times r_{i_k i_1} > 1.$$

Devise an efficient algorithm that finds such an anomaly if one exists. Justify the correctness of your algorithm and its runtime.

6. **How is the course so far?** (0.5 points extra credit)

Please complete the poll at `https://goo.gl/forms/G6VGBMAqJa41IdKv2`.