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 points) 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]**

**SOLUTION:**

(a) Assume that surfboard \( j \) is used in this optimal allocation, but that it is not allocated to surfer \( i \). Instead, assume that surfboard \( j \) is allocated to surfer \( i' \), and some other surfboard \( j' \) is assigned to surfer \( i \).

Consider two allocations:
i. $f$: the optimal allocation, where $f(i) = j'$ and $f(j) = i'$
ii. $f'$: allocation identical to $f$, but swapping the assignments of surfers $i$ and $i'$, so $f'(i) = i'$ and $f'(j) = j'$

The cost of allocation $f$ is $A = |h_i - s_{j'}| + |h_{i'} - s_j| + C$
The cost of allocation $f'$ is $A' = |h_i - s_j| + |h_{i'} - s_{j'}| + C$
where $C$ is the cost of all other matches.

Note that, since allocation $f$ is optimal, then $A \leq A'$. We want to show that $A' \leq A$ to get that $A' = A$.

Consider 6 cases of orderings of $h_i, h_{i'}, s_j, s_{j'}$. By the ordering of surfer heights and surfboard lengths, we already know that that $h_{i'} < h_i$ and $s_j < s_{j'}$, so this reduces the total number of orderings from 24 to $24/4 = 6$.

Case 1: $s_{j'} < s_j < h_{i'} < h_i$
Case 2: $s_{j'} < h_{i'} < s_j < h_i$
Case 3: $s_{j'} < h_{i'} < h_i < s_j$
Case 4: $h_{i'} < s_{j'} < h_i < s_j$
Case 5: $h_{i'} < s_{j'} < s_j < h_i$
Case 6: $h_{i'} < h_i < s_{j'} < s_j$

Cases 2-5 contradict the optimality of $f$ so can’t be true.

Consider case 1. We have that
\[
A = |h_i - s_{j'}| + |h_{i'} - s_j| + C \\
= h_i - s_{j'} + h_{i'} - s_j + C \\
= h_i - s_j + h_{i'} - s_{j'} + C \\
= |h_i - s_j| + |h_{i'} - s_{j'}| + C \\
= A'
\] (1)

Consider case 6. We have that
\[
A = |h_i - s_{j'}| + |h_{i'} - s_j| + C \\
= -(h_i - s_{j'}) - (h_{i'} - s_j) + C \\
= (-h_i + s_{j'}) - (h_{i'} - s_j) + C \\
= (h_i - s_j) + (h_{i'} - s_{j'}) + C \\
= A'
\] (2)

(b) $A[i, j]$ corresponds to the optimal allocation of the first $j$ surfboards to the first $i$ surfers.

If $j$ is not used in this optimal allocation, then $A[i, j] = A[i, j - 1]$.
If $j$ is used in the optimal allocation, then there is an optimal allocation where it is allocated to surfer $i$.

Overall, $A[i, j] = \min(A[i, j - 1], A[i - 1, j - 1] + |h_i - s_j|)$
(c) The DP algorithm is based on the previous recurrence relation. We keep track of a 2-dimensional table, each entry of the table corresponds to a state \((i, j)\) for \(1 \leq i \leq n\) and \(1 \leq j \leq m\). We start at the bottom right corner \((i = n, j = m)\). For each \((i, j)\), we compute (recursively) \(A[i - 1, j - 1]\) and \(A[i - 1, j - 1] + |h_i - s_j|\) and we take the minimum of the two quantities.

If \(A[i - 1, j - 1] + |h_i - s_j|\) is the minimum, then we keep track that \(f(i) = j\). Once we have reached the state \((i = 1, j = 1)\), we return the values \(f(1), ..., f(n)\).

(d) (1 point) The table has \(mn\) elements, each corresponding to a state \(A[i, j]\). But, note that once we reach \(A[i, i]\), we don’t have to recurse anymore because it is trivial how to allocate the first \(i\) surfboards to the first \(i\) surfers: give them by order of height. So, we don’t have to compute the states that are in the bottom left corner (i.e states \((i, j)\) for \(1 \leq i \leq n\) and \(1 \leq j \leq n\)) and in the top left corner (i.e. states \((i, j)\) for \(1 \leq i \leq n\) and \(m - n \leq j \leq m\)). So, we only have to compute \(n(m - n)\) states. Since computing each state takes only constant time, the total running time is \(O(mn)\).

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 \(A[r_u, 1], A[r_u, 2], ..., A[\ell_u, 1], A[\ell_u, 2], ...\). [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.]

**SOLUTION:**

(a) Let the weight of node $u$ be $w_u$. $A[u,i] = w_u + \max_{0 \leq j \leq i-1}\{A[r_u,j] + A[l_u, i-1-j]\}$.

(b) Let $T(u)$ be the number of nodes in the subtree rooted at any node $u$. $T(u)$ can be computed by traversing the tree in $O(n)$ time. The program starts at $A[r,k]$, where $r$ is the root node. It computes $A[u,i]$ for $1 \leq u \leq n$ and $1 \leq i \leq k$:

$$A[u,i] = \begin{cases} 0 & \text{if } i = 0 \\ -\infty & \text{if } i > T(u) \\ w_u + \max_{0 \leq j \leq i-1}\{A[r_u,j] + A[l_u, i-1-j]\} & \text{otherwise} \end{cases}$$

We store $j$ which maximizes sum of weights for $A[u,i]$ in $B[u,i]$. Using $B$, the program can output the sizes of left subtree and right subtree rooted at each node in the optimal tree, starting from root node $r$. The program also outputs $A[r,k]$.

For any node $u$ with exactly $i$ nodes in the subtree rooted at $u$, suppose we correctly computed $A[r_u,j] + A[l_u, i-1-j]$ for all $0 \leq j \leq i-1$. The maximum weight must be the weight of $u$ plus the maximum sum of weights from the left subtree and right subtree of $u$. So the program can output the optimal subtree rooted at any node.

(c) The program computes $T(u)$ for each node $u$ at the beginning in $O(n)$. In the program, we construct a table of $n \times k$ elements, each one corresponding to $A[u,i]$. For each entry $A[u,i]$, we need to compute the max of $O(k)$ sums. Thus, the time complexity is $O(nk^2)$.

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.)

**SOLUTION:**

(a) We are interested in maximizing the value on day 2 of the stocks that we purchase on day 1. Given a budget of \( P \), we can compute the maximum profit, \( M[l] \) attainable over the \( k \) stocks while spending \( l \) dollars, for \( l \in [1, P] \). Note that for a given \( l \),

\[
M[l] = \max_{j \in [0,k], p_1,j \leq l} \left( (p_2,j - p_1,j) + M[l - p_1,j] \right)
\]

That is, if we have \( l \) dollars to spend, we can maximize our profits by considering the maximum profit attainable by purchasing a unit of the \( j \)th stock as well as the maximum profit attainable with a budget of \( l \) minus the price of the \( j \)th stock.

We can maintain a \( 1 \times P + 1 \) array \( M \) of profits for each price \( l \in [0, P] \). Start with \( M[0] = 0 \). Then, for each \( l \), assuming \( M[l'] \) is computed correctly for all \( l' \leq l \), we can compute \( M[l] \) according to the equation above by iterating over the \( k + 1 \) stocks.

Finally, we will pass over \( M \) and look for the entry of maximum value - this is the maximum attainable profit by investing the money available.

The running time for the algorithm will be the time it takes to iterate over each of \( O(P) \) budget allowances, and at each budget allowance over each of \( O(k) \) stocks, along with one more pass over the array. Thus, it is \( O(kP) \).

(b) To find the optimal investment strategy over \( n \) days, consider applying our strategy for two days at each interval. If we ever discover that our earnings exceed \( Q \) dollars, we can return success immediately. Otherwise, we have to continue running our algorithm to compute the profits attainable from day \( i \) to day \( i + 1 \) (and add these profits to our starting budget) to obtain our budget for the \( i + 1 \)th day. If after \( n \) days, we never obtained \( Q \) dollars, we can report the maximum earnings we can make, by summing \( P \) and the total maximum profits.

To see why this procedure will produce the optimal profits, suppose that in the optimal investment strategy, we purchase a stock on day \( i \) and sell it on day \( j \). Because we are allowed to buy and sell as many stocks as we want per day without cost, this is equivalent to purchasing the stock on day \( i \), selling it on day \( i + 1 \), then purchasing on day \( i + 1 \) and selling on day \( i + 2 \), etc. until we buy on day \( j - 1 \) and sell on day \( j \). Thus, at each step, we only need to maximize the total for the next day.

To see why our runtime will be \( O(nkQ) \), note that each of our budgets before success will be some \( P < Q \). Thus, we can bound the running time of each of the \( n-1 \) iterations by \( O(kQ) \). Thus the overall runtime will be \( O(nkQ) \).

4. **Toll Game.** (5 points)

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, you 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:

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

Here \( w(p) \) denotes the path length, \( w(x_0, \ldots, x_k) = \sum_{j=1}^{k} w(x_{j-1}, x_j) \), 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.]

**SOLUTION:**

Let \( \lambda \) be an arbitrary real number. The modified edge weights \( w_{\lambda}(u, v) = w(u, v) - \lambda \) correspond to the case where we receive an additional sum of \( \lambda \) dollars at each step. This simply reduces the average cost by \( \lambda \), i.e., the minimum average cost is now equal to \( \lambda_* - \lambda \).

If there is a reachable negative weight cycle for the modified weights, we can traverse it *ad infinitum*, gaining money in the long run. (The weights of initial non-cycle edges on the path contribute less and less as the number of moves goes to infinity.) Therefore, in this case the asymptotic average cost is negative. Otherwise the cost of any path is bounded from below, hence the asymptotic average cost is nonnegative. Thus, \( \lambda_* - \lambda \geq 0 \) if and only if there is no reachable negative-weight cycle with respect to the weight system \( w_{\lambda} \).

We need to represent this condition in a more manageable form.

Let us follow the hint and use these functions:

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

**Theorem.** The graph contains no reachable negative-weight cycle if and only if \( \forall x \tilde{g}_{\lambda, n}(x) = \tilde{g}_{\lambda, n-1}(x) \).
Here, the weight system \( w_\lambda(u, v) = w(u, v) - \lambda \) is assumed. (We only proved the “only if” part in class, but the “if” part is also easy.) Now we need to figure out how \( \tilde{g}_{\lambda,k}(x) \) depends on \( \lambda \); that’s where the function \( g_k \) comes in useful:

\[
\tilde{g}_{\lambda,k}(x) = \min_{0 \leq j \leq k} (g_j(x) - j\lambda)
\]

Hence the condition in the above theorem can be rewritten as follows:

\[
\tilde{g}_{\lambda,n}(x) = \tilde{g}_{\lambda,n-1}(x) \iff \exists j < n \, g_j(x) - j\lambda \leq g_n(x) - n\lambda \iff \lambda \leq \max_{0 \leq j \leq n-1} \frac{g_n(x) - g_j(x)}{n-j},
\]

where we assume that \( g_n(x) < \infty \). Thus,

\[
\lambda_* = \max\{\lambda : \text{there is no reachable negative-weight cycle w.r.t. the weight system } w_\lambda\} = \min_{x \in A} \max_{0 \leq j \leq n-1} \frac{g_n(x) - g_j(x)}{n-j},
\]

where \( A \) is the set of vertices that are reachable from \( s \).

Computing \( g_j(x) \) for \( j = 0, \ldots, n \) using dynamic programming takes \( O(nm) \) time. The subsequent computation of \( \lambda_* \) using above equation is an \( O(n|A|) = O(nm) \) task. Thus, the total running time is \( O(nm) \).

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

\[
r_{i_1i_2} \times r_{i_2i_3} \times \cdots \times r_{i_{k-1}i_k} \times r_{i_ki_1} > 1.
\]

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

7
SOLUTION:

(a) (2 points) **Algorithm:** Build $G = (V,E,w)$ as in part (a), but with weights $w(C_i, C_j) = -\log(r_{ij})$. Run Bellman-Ford from $C_s$ and return a shortest $C_s \to C_t$ path.

**Correctness:** As in part (a), a sequence of exchanges corresponds to a path in $G$. However, here we want a path $C_{i_1} = C_s, C_{i_2}, \ldots, C_{i_k} = C_t$ that maximizes $\prod_{\ell=1}^{k-1} r_{i_{\ell},i_{\ell+1}}$. Since log is a monotonically increasing function (i.e. if $a \geq b$, then $\log(a) \geq \log(b)$), this is the same as maximizing $\log \left( \prod_{\ell=1}^{k-1} r_{i_{\ell},i_{\ell+1}} \right) = \sum_{\ell=1}^{k-1} \log(r_{i_{\ell},i_{\ell+1}})$. Finally, this is equivalent to minimizing $\sum_{\ell=1}^{k-1} -\log(r_{i_{\ell},i_{\ell+1}}) = \sum_{\ell=1}^{k-1} w(C_{i_{\ell}}, C_{i_{\ell+1}})$, which is the shortest path objective. Note that we must use Bellman-Ford rather than Dijkstra’s algorithm, since these weights may be negative.

**Running time:** $O(n^3)$ total. $G$ can be built in time $O(n^2)$ time, and Bellman-Ford takes $O(n^3)$ time since we have $\Theta(n^2)$ edges.

(b) (2 points) **Algorithm:** Run Bellman-Ford on the same graph as in part (b); then execute one more iteration of Bellman-Ford to check if there is a negative cycle in $G$. If there is, the cycle is the anomaly—trading currencies along the cycle will result in a profit.

**Correctness:** A currency anomaly $\prod_{\ell=1}^{k-1} r_{i_{\ell},i_{\ell+1}} > 1$ implies (by the same log manipulations we did in part (b)) that $\sum_{\ell=1}^{k-1} w(C_{i_{\ell}}, C_{i_{\ell+1}}) = \sum_{\ell=1}^{k-1} -\log(r_{i_{\ell},i_{\ell+1}}) < 0$. Thus there is a negative cycle in $G$, which can be found by an extra iteration of Bellman-Ford, as shown in lecture 11. **Running time:** $O(n^3)$ total. We are doing the same thing as in part (b), plus one extra iteration which takes $O(|E|) = O(n^2)$ time.

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

Please complete the poll at https://goo.gl/forms/rV8iC1ihWext04oF2.