Homework 4: Dynamic Programming
Due: October 12, 2023

Problem 1. A company is shipping inventory with the help of a shipping container. The company has reserved two slots in the container with heights $H_1$ and $H_2$, respectively, and they can only package their $n$ pieces of inventory vertically.

You are given the heights $h_i$ and values $v_i$ for each item, and you are tasked with calculating the maximum value of inventory the company can ship.

(a) Design an algorithm that determines the maximum value in $O(H_1 H_2 n)$ time and space.

(b) Provide a proof of correctness of your algorithm.

(c) Justify your algorithm’s runtime and memory utilization.

Solution. This problem is very related to the 0/1 Knapsack problem presented in class.

Algorithm 1 Double Knapsack

Input: an array $h$ containing the item heights,
     an array $v$ containing the item values,
     the height $H_1$ of the first slot,
     the height $H_2$ of the second slot

Output: The maximum value of inventory that can be stored in the container

1: function MaxInventory($h$, $v$, $H_1$, $H_2$)
2:    $n \leftarrow$ the length of $h$ \hspace{1cm} $\triangleright$ this is equal to the length of $v$
3:    Initialize ‘dp’ to be a new 3D array with dimensions $n$, $H_1$, and $H_2$
4:    for $i$ from 0 until $n$ do
5:        for $h_1$ from 0 until $H_1$ do
6:            for $h_2$ from 0 until $H_2$ do
7:                if $i = 0$ then
8:                    $dp[i][h_1][h_2] = v[i]$
9:                else \hspace{1cm} $\triangleright$ assume $h[i] \leq h_1$, $h_2$
10:                   $temp \leftarrow$ Max($dp[i-1][h_1-h[i]][h_2]$, $dp[i-1][h_1][h_2-h[i]]$)
11:                   $dp[i][h_1][h_2] =$ Max($v[i] + temp$, $dp[i-1][h_1][h_2]$)
12:    return $dp[n-1][H_1-1][H_2-1]$ \hspace{1cm} $\triangleright$ 0-indexing

... 

Proof of Correctness:
We’ll induct over the predicate $P(k)$: our algorithm correctly computes the maximum value of the first $k$ items for all valid height pairs in $[H_1] \times [H_2]$. 

Base Case: $P(1)$ holds because our algorithm sets the maximum value for all height pairs to $v[0]$ when $i == 0$.

Inductive Step: For $k \geq 1$, suppose that $P(k)$ holds. For any valid height pair $(h_1, h_2)$, we can add the $(k+1)th$ item to the first slot or the second slot (not inclusive) and add $v[k]$ to the maximum value achieved on the resulting height pairs. Alternatively, we can disregard the $(k+1)th$ item. Our recursive reasoning holds because $dp[k-1]::$ is correctly filled according to the inductive hypothesis. Since our algorithm handles this step in lines 10-11, $P(k+1)$ must hold.

Since our base case and inductive step hold, $P(k)$ holds for all non-negative $k$.

**Time Complexity**: Our algorithm fills every cell in the dp tensor and retrieves, at most, 3 previously computed cells from dp. Thus, its runtime is $O(H_1 H_2 n)$.

**Memory Utilization**: The memory bottleneck of our algorithm is the dp tensor, which has dimensions $H_1 \times H_2 \times n$. Each cell uses $O(1)$ space, so our memory utilization is $O(H_1 H_2 n)$. 
Problem 2. You are given a roll (array) of \( n \) magical coins. Each coin has an initial value \( c_i \) which increases at each time step linearly. Therefore, at time step \( t_j \) the coin has a value of \( c_{i,j} \).

You want to sell the coins to maximize your total profit, however, you are only able to sell the coins one at a time. Furthermore, you are only able to sell the coins from either end of the roll of coins.

(a) You initially spring for a greedy approach to sell your magical coins, i.e. at each time step you compare both ends of the array and sell the coin with least value. Is this strategy flawed? Explain your reasoning.

(b) Design an algorithm which determines the optimal order to sell the magical coins to maximize profit, given that you know the starting value of each coin. Your algorithm should run in \( O(n^2) \) time. Provide a proof of correctness for your algorithm, and justify its runtime and memory utilization.

Solution.

(a) As a counterexample, consider a roll with the following coin values: 4, 3, 3, 1, 1, 5. The greedy approach would yield a profit of 58 despite the fact that we can do better by removing all the coins from the right, which would achieve a profit of 61.

(b)

\begin{verbatim}
Input: A list of integers representing the values of each coin
Output: The optimal selling order of the coins
1: function OPTIMALSELLINGORDER(coins)
2: Initialize ‘dp’ to be a new 2D array with dimensions \( n \) and \( n 
3: for i from 0 until n-1 do ▷ consider all subarray lengths
4:   for j from 0 until n-i-1 do
5:     if i == 0 then
6:       dp[j][j+i] = coins[j]*(n-i)
7:     else
8:       dp[j][j+i] = Max((n-i)*c[j]+dp[j+1][j+i], ((n-i)*c[j+i])+dp[j][j+i-1])
9: return dp[0][n-1]
\end{verbatim}

Proof of Correctness: We’ll induct over the predicate \( P(k) \) : our algorithm correctly computes the maximum total profit for all rolls of length \( k \).

Base Case: \( P(1) \) holds because the maximum profit we can make with a single coin is equal to its value multiplied by the appropriate timestamp. This is handled in line 6.

Inductive Step: Suppose \( P(k) \) holds for some non-negative \( k \). The maximum profit for any subarray of length \( k+1 \) can be achieved by selling the left coin or right coin first and maximizing the profit
among the remaining $k$ coins to the left or right of it, respectively. This logic is handled in line 8 of our algorithm, and the recursive relationship holds because our inductive hypothesis holds, proving $P(k + 1)$.

Since our base case and inductive step both hold, $P(k)$ holds for all non-negative $k$.

**Time Complexity**: Our algorithm fills every cell of the dp matrix in $O(1)$ time, so it runs in $O(n^2)$.

**Memory Utilization**: The memory bottleneck of our algorithm is the dp matrix, which uses $O(n^2)$ space. Thus, the algorithm uses $O(n^2)$ space.
Problem 3. Given an *unsorted* array of $n$ positive integers, your task is to remove the least number of elements from the start or end of the array until twice the minimum of the array is larger than the maximum.

(a) Design a dynamic programming algorithm which runs in $O(n^2)$ time. Provide a proof of correctness for your algorithm, and justify its runtime and memory utilization.

(b) Suppose the integers are sorted in non-decreasing order. Design an algorithm that runs in $O(n)$ time. Provide a proof of correctness for your algorithm, and justify its runtime and memory utilization.

Solution. Input: Array of $n$ positive integers
Output: Least number of elements removed

```
1: function REMOVELEASTELEMENTS(arr, n)
 2:   output ← $n - 1$
 3:   dp ← ARRAY[n][n][2] ▷ each cell stores min and max num in subarray
 4:   for i from 0 until $n - 1$ do
 5:     for j from 0 until $n - i - 1$ do
 6:       if $i == 1$ then
 7:         dp[j][j + i][0], dp[j][j + i][1] ← arr[j]
 8:       if $i ≠ 1$ then
 9:         dp[j][j + i][0] ← min(dp[j + 1][j + i][0], dp[j][j + i - 1][0])
10:        dp[j][j + i][1] ← max(dp[j + 1][j + i][1], dp[j][j + i - 1][1])
11:       if $(2 * dp[j][j + i][0]) > dp[j][j + i][1]$ then
12:          output ← min($n - i - 1$, output)
13:   return output
```

1. Proof of Correctness:
Our algorithm uses dynamic programming to calculate the min / max of each subarray. If both are computed correctly for each subarray, output must be the minimum number of elements that we need to remove to satisfy the condition where twice the min is greater than the max.

To prove that each subarray stores the correct min / max, we’ll use induction over the predicate $P(k)$: our algorithm correctly computes the min / max of all sub-arrays of length $k$.

Base Case: $P(0)$ holds because a single element is both the minimum and maximum.

Inductive Step: Suppose $P(k)$ holds for some non-negative $k$. For a subarray of length $k + 1$ starting at any valid index $j ∈ 0, . . . , n - k - 1$, the minimum of the overall subarray will be the minimum of the $k$ elements to the right of $j$ and the $k$ elements to the right of
index \( j + k \). The similar logic holds for the maximum. Our inductive hypothesis guarantees us that these \( k \)-length subarrays have their minimums and maximums correctly computed, which proves that \( P(k + 1) \) holds.

Since our base case and inductive step hold, \( P(k) \) holds for all non-negative \( k \), which proves that our overall algorithm is correct.

**Time Complexity:** Our algorithm performs a constant time computation for each of the \( O(n^2) \) entries in the \( dp \) tensor, which means that its runtime is \( O(n^2) \).

**Memory Utilization:** The memory bottleneck of the algorithm is the \( dp \) tensor. Since each cell holds a constant amount of space, the overall memory utilization of the algorithm is \( O(n^2) \).

2. Description: our algorithm uses a sliding window to find the largest subarray that satisfies the condition that twice the smallest element is greater than the largest element. By default, we initialize a left pointer, \( l \), to point to \( arr[0] \) and set a right pointer, \( r \), to point to the largest number satisfying our condition (w/ \( arr[0] \) as the minimum). While both pointers are in the bounds of the array, we consider the following cases:

(a) If the subarray satisfies the aforementioned condition (using the elements at the left and right pointers), we update a global minimum variable with the number of elements we need to remove to get that subarray. We then move the right pointer right.

(b) Otherwise, we move the left pointer right.

**Proof of Correctness:** we prove that our algorithm is correct by showing that it never prunes the optimal solution (ie. the largest subarray satisfying the condition):

(a) If a subarray satisfies the condition, we can only get a larger subarray that is also valid by moving the right pointer right. Moving the right pointer left or the left pointer right would only yield a smaller (sub-optimal) subarray. Moving the left pointer left would bring us back to a subarray that we’ve previously considered.

(b) Otherwise, the subarray does not satisfy the condition, so we can only get a valid subarray by moving the left pointer right. Moving the right or left pointer left would bring us back to previously considered subarrays. Moreover, moving the right pointer right can only increase the subarray’s maximum element, which cannot possibly make it valid.

Since the largest subarray satisfying the condition is never pruned, our algorithm is guaranteed to return an optimal solution.

**Time Complexity:** In every iteration of our loop, we either move the left or right pointer right and perform some constant time operations. Since the left and right pointers traverse \( O(n) \) elements each, the loop performs \( O(2n) = O(n) \) iterations.

**Memory Utilization:** We use a constant amount of space for the global minimum variable (the output) and the left / right pointers. Thus, we use \( O(1) \) memory.