Solution 4: Dynamic Programming

Problem 1. 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)

```
Input: A list of integers representing the values of each coin
Output: The optimal selling order of the coins
function OptimalSellingOrder(coins)
    Initialize ``dp'' to be a new 2D array with dimensions $n$ and $n$
    for $i$ from 0 until $n-1$ do                    ▷ consider all subarray lengths
        for $j$ from 0 until $n-i-1$ do
            if $i == 0$ then
                $dp[j][j+i] =$ coins$[j] * (n-i)$
            else
                $dp[j][j+i] =$ Max$(((n-i)*c[j]) + dp[j+1][j+i],\ ((n-i)*c[j+i]) + dp[j][j+i-1])$
            end if
        end for
    end for
    return $dp[0][n-1]$
end function
```

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.

$\square$

Problem 2. 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. (a) Consider the following algorithm:

```
Input: Array of $n$ positive integers
Output: Least number of elements removed
function RemoveLeastElements($arr, n$)
    output $\leftarrow n - 1$
    $dp$ $\leftarrow$ ARRAY[$n$][$n$][2] ▷ each cell stores min and max num in subarray
    for $i$ from 0 until $n - 1$ do
        for $j$ from 0 until $n - i - 1$ do
            if $i == 1$ then
                $dp[j][j+i][0], dp[j][j+i][1] \leftarrow arr[j]$
            end if
            if $i \neq 1$ then
                $dp[j][j+i][0] \leftarrow \min(dp[j+1][j+i][0], dp[j][j+i-1][0])$
                $dp[j][j+i][1] \leftarrow \max(dp[j+1][j+i][1], dp[j][j+i-1][1])$
                if $(2 * dp[j][j+i][0]) > dp[j][j+i][1]$ then
                    output $\leftarrow \min(n - i - 1,$ output$)$
                end if
            end if
        end for
    end forreturn output
end function
```

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 $j \in 0, \ldots, n-k-1$, 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 $O(n^2)$.

(b) **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.

□

Problem 3. You are in charge of a botanical garden, and you're designing a pathway where visitors can admire a sequence of unique flower beds arranged in strictly increasing order of their charm. Each flower bed currently has a charm rating stored in the array A.

There is a problem; the current order of these ratings is not strictly increasing, which could disrupt the visitor experience. To correct this, you have access to a nursery full of replacement flowers, each with its own charm rating stored in another array, B. In one operation, you can choose any flower bed along the path and replace its charm rating with one from the nursery. It is guaranteed that $|B| \geq |A|$.

(a) Design a **dynamic programming** algorithm that determines the minimum number of operations to ensure the charm ratings in A are strictly increasing.

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

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

Solution. (a) Consider the following algorithm:

Algorithm 1 StrictlyIncreasingArr

```
Input: A, B
Output: The minimum number of operations needed to ensure that A is strictly
   increasing
function MAKEINCREASING($arr1$, $arr2$)
   sort B
   dp $= [\{\}] * |A|$
   dp[0][A[0]] = 0;
   dp[0][B[0]] = 1;
   for i in range(1, |A|) do
      newdict = {}
      for prev in dp[i - 1] do
         if $A[i]$ > $prev$ then
            dp[i][A[i]] = min (dp[i][A[i]], dp[i - 1][prev]
         end if
         k = bisectright(B, prev)
         if k < len(B) then
            dp[i][B[k]] = min(dp[i][B[k]], 1 + dp[i - 1][prev])
         end if
      end for
   end forreturn min(dp[|A| - 1].values) if dp else -1
end function
```

(b) The general idea of the algorithm is to build the possible solutions as we iterate over A, and consider swapping/keeping at each step. Each $dp[i][prev]$ will store the minimum number of swaps needed to sort the first $i + 1$ flowers of A such that the $A[i] = prev$.

We induct on i to prove that dp stores exactly what we want it to. The base case is either we keep the first element, in which case there are 0 swaps, or we make one swap. In this case, it only makes sense to swap with the smallest element of B, which we know to be $B[0]$ after sorting. Suppose the algorithm holds after $i = k$ iterations. On the $k + 1$st iteration, consider each value $prev \in dp[k]$. If $A[k + 1] > prev$, we know that a swap is not needed as we are guaranteed the first k elements are sorted. Thus, we can achieve sorting the first $k+1$ elements in $dp[k][prev]$ swaps, which is what we have in line 10.

The next case to consider is swapping: if we are to swap, we need to swap with an element that has value greater than $prev$. It only makes sense to swap with the smallest element in B that is greater than $prev$: this is precisely binary search! In our algorithm, we use Python's bisect library to achieve this. In this case, getting the first $k + 1$ elements sorted with $A[k + 1]$ as our new value from B takes $dp[k][prev] + 1$ swaps, which is reflected in line 14. Therefore, the given form of the algorithm correctly computes the minimum number of swaps needed to reach all the states we are considering.

To finish, we know that $dp[|A| - 1]$ holds the minimum number of swaps to sort the first $|A|$ values with a fixed value of $A[|A| - 1]$. We seek the minimum over all these values, which corresponds to $\min(dp[|A| - 1].values)$, as desired.

(c) The initial sorting of B runs in $O(|B| \log |B|)$. At each step, we iterate through the dictionary's keys, of which there are at most $|B| + 1$ (the value of $prev$ can only be an element of B or the array element itself. At each step, we perform constant time computations and one binary search on B, so the runtime of the main logic is $O(|A||B| \log |B|)$. Thus, the runtime is

$$O(|B| \log |B|) + O(|A||B| \log |B|) \in O(|A||B| \log |B|)$$

The memory utilization is $|A|$ dictionaries of size at most $|B| + 1$, as mentioned in the previous paragraph. This gives a memory usage of $O(|A||B|)$.

□