Homework 4

Solution Key

This homework must be typed in LaTeX and handed in via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

Problem 1

Tianren *really* likes **reversible** words, that is words that read the same forwards and backwards (e.g, racecar, kayak, level). He likes them so much, that he has started to try and find reversible words in any strings that he sees.

1. Given a string with n characters from the English alphabet (you may assume that they are lower case), design an efficient dynamic program that finds the longest substring (a consecutive sequence of characters) of the input string containing a *reversible* word.
2. Provide proof of the correctness of the algorithm and analyze its time complexity and memory utilization.

Algorithm: string s with n characters is a palindrome if $p[0] = p[n-1]$ and $p[1, n-2]$ is a palindrome. We can construct an $n \times n$ matrix A that indicates at each $A[i][j]$ whether $s[i, j]$ is a palindrome. Note that we cannot take a substring where the starting index is greater than ending index, so we will only consider cases where $i \leq j$.

- **Base Case 1:** A one character string is always palindromic. Therefore, if $i = j$, then $A[i][j] = 1$.
- **Base Case 2:** A two character string is palindromic if the first character and the second character are equal. Therefore, if $j - i = 1$, $A[i][j] = 1$ if $s[i] = s[j]$ and $A[i][j] = 0$ if $s[i] \neq s[j]$.
- **General Case:** In general, if $A[i+1][j-1]$ is a palindrome, then $A[i][j] = 1$ if $s[i] = s[j]$. However, if $A[i+1][j-1] = 0$ or $s[i] \neq s[j]$, then $A[i][j] = 0$.

As we iterate all relevant indices of A, we can keep track of the starting and ending indices of the longest palindromic substring so far.

Correctness: We argue that $A[i][j]$ correctly indicates whether $s[i, j]$ is a palindrome.

- Trivially, base case 1 is correct, as we will produce the same string by reversing a one character string.
- Similarly, base case 2 is correct as a two character string is palindromic if and only if its only two characters are the same.

- Lastly, the general case is correct because palindromes are mirrored by definition. If an arbitrary string is palindromic and we tack character x to its beginning and end, the resulting string will also be palindromic. Therefore, $A[i][j]$ will only equal 1 if the internal string is palindromic (given by $A[i+1][j-1]$) and its first and last characters are equal– which, by definition, makes $A[i][j]$ a palindrome.

It follows that if we maintain some variable `max_palindrome` and compare / update it wherever $A[i][j] = 1$, we will have checked all palindromes of s and recorded the longest one.

Runtime: We iterate through A, an $n \times n$ matrix, while keeping track of the longest palindromic substring. Filling in each $A[i][j]$ is a constant time operation. Therefore, the algorithm is $O(n^2)$.

Problem 2

In his dorm room, Hammad has a box of textbooks, $B = \{b_1, b_2, \ldots, b_n\}$, each with corresponding weight w_i, which is a positive integer. When Hammad moves back home at the end of the semester, he wants to bring as many pounds of books home as possible. However, he has a problem: the moving company can only move at most W pounds of books, so Hammad will have to decide which books to take with him and which to leave behind.

Design an algorithm to help Hammad find maximum weight of books he can bring back that does **not** exceed the weight limit of the moving company (W). Your algorithm should have runtime $O(nW)$.

Provide proof of the correctness of the algorithm and analyze its time complexity and memory utilization.

Hint: Each element b_i is either in or out of the maximal sum.

Algorithm: Start with the first item, and consider a set S of all of the possible valid subset sums up to W. Pseudocode for this idea is given as follows:

Algorithm 1 Subset Sum to a Threshold

```
1: Instantiate a set S_1 with values {0, w_1}
2: for 2 ≤ i ≤ n do
3:     S_i ← Empty Set
4:     for s_j ∈ S_{i-1} do
5:         Add s_j to S_i
6:         if s_j + w_i ≤ W then
7:             Add s_j + w_i to S_i
8: return max(S_n)
```

Correctness: We will first prove that S_n contains all valid sums of subsets less than W via induction on i (the iteration variable).

Base Case ($i = 2$): We have to consider the subsets $\{\}, \{a_1\}, \{a_2\}, \{a_1, a_2\}$. We note that two of them are in S_i, and therefore by line 5 are in S_2, and the others are included by the condition included in line 7 (rejected if the sum exceeds W, which is what we expect). Therefore, in this case S_2 will include all of the valid subset sums.

Inductive Hypothesis ($i = k$): Assume that S_k contains all valid subset sums of the partial set $B_k = \{b_1, \ldots, b_k\}$.

Inductive case ($i = k + 1$): By the inductive hypothesis, we assume that S_k contains all of the valid subset sums of subsets of B_k less than W. Then, by the same argument as the base case (lines 5-7 of the pseudocode), we consider all subsets $B' \subset B_k$ and $B' \cup \{b_{k+1}\}$ (by taking both s_j and $s_j + w_k$, we consider all possible subset sums). Further, we know that when we reject a sum because it exceeds W, since w_k is a positive integer, we know it will never become valid after it is rejected, which we do properly.

Therefore, since this means that S_n contains all possible valid subset sums (less than W), and in the last step we take the maximum, that this must be correct (by definition).

Time Complexity: We note that for each step i, we read $|S_{i-1}|$ elements from S_{i-1} and then add at most $2\,|S_{i-1}|$ elements to S_i. However, since S_k is a set there are no multiples of identical elements, and we only add elements that are at most W, meaning that since w_i is a positive integer, we can see that $|S_i| \leq W$ for any i.

Thus, each step of the outer loop requires $O(W)$ operations, and since we perform $n-1$ of these loops (and then perform a max operation taking n operations), our overall runtime is $\boxed{O(nW)}$.

Memory Utilization: At any given time, we only have to maintain S_{i-1} and S_i in memory, which means that the utilization is bounded by $2C$ (since $|S_j| \leq C$). Therefore, since the only other storage required are temporary variables, the memory utilization of this algorithm is $\boxed{O(W)}$.

Problem 3

Rohit gave Rob a case of N magical coins arranged in an array. Each coin has an initial value (C_i) which increases with every time-step (t_j). Time-step starts at 1 ($t_1 = 1$) and the value of any coin (C_i) at time (t_j) would be $C_i \cdot t_j$, i.e., the product of the initial value of the coin and the current time-step.

Rob wants to sell the coins but he can only sell one coin at a time. Also, at any time-step he can only sell a coin from the *start* of the array or the *end* of the array otherwise he would have to bear the curse of grading 1000 assignments (we really don't want Rob to be cursed).

1. Help Rob find the maximum profit that he can make by giving him an algorithm that runs in $O(N^2)$. Prove correctness of your algorithm, and justify its runtime and memory utilization.
2. Rob decided to go greedy by selling the ***smallest*** coin from either end at any given timestep, thereby decreasing the time complexity to $O(N)$. Is this strategy flawed? Explain why or why not.

1. **Solution** We will need to check the cost for every possible configuration and store the states in a matrix so that we don't visit the same state multiple times. We can use the below mentioned pseudo-code to implement this. We pass the array of coins to the function MaxProfit.

Algorithm 2

```
dp ← []
procedure MaxProfit(L, R, coins)  ▷ Initial value of L is 0 and R is N − 1
    if L > R then
        return 0
    if dp[L][R] ≠ NULL then                    ▷ If the state is already visited
        return dp[L][R]
    year ← (N − (R − L))
    lsum ← coins[L]· year + MaxProfit(L + 1, R, coins)
    rsum ← coins[L]· year + MaxProfit(L, R − 1, coins)
    dp[L][R] ← max(lsum, rsum)
    return dp[L][R]
```

Note: The above algorithm is written recursively with memoization, but it could just as easily be written iteratively over L, R. Both of these approaches are equivalent.

Correctness: Correctness follows inductively from the fact that for base cases where $L > R$, we know that these can't exist, so we return 0.

Otherwise, for each combination of L, R, we consider the maximum value of the two steps that could have come before (either $(L+1, R)$ or $(L, R-1)$), which is the best value that we could have done reaching the current point. We note that by iterating this process, we consider every possible relevant combination and ordering for the choices we make.

Therefore, by induction on L, R using the argument from above, this algorithm must be correct.

Runtime: We compute each entry $dp[L][R]$ once, and each time do $O(1)$ operations, meaning that our overall runtime is going to be $O(N^2)$.

Memory Utilization: We note that dp is of size N^2, and in each recursive step we only require 3 temporary variables (which could be $3N$ in the worst case, for every stack), therefore the memory utilization is $O(N^2)$.

2. **Solution** The greedy approach would not work. This is because we are constrained to pick coins from only the two end-points of the array. If we greedily picked the greatest/smallest value at every step, there can be a bigger value that can be accessed at a later stage by selecting the other value. This can be easily shown by an example. If coins = [2,4,6,2,5] and we greedily selected the smallest value each time, the final profit will be 63. But if we used dynamic programming and checked all the possible solutions, the final profit will be 64.