CS 577: Introduction to Algorithms	Homework 5 Solutions

Problem 1

(a)

For $j \geq i$, define $\mathsf{MaxValue}(i, j)$ as the maximum value possible achieved using the subsequence of numbers $a_i \ldots a_j$ and operators $o_{i,i+1} \ldots o_{j-1,j}$. In the case that $i = j$, there is no operator and only the number a_i. Similarly, define $\mathsf{MinValue}(i, j)$ as the minimum value possible. Recall that for dynamic programming we will be computing these values iteratively (not recursively) by using a memoization matrix. So we will initialize an $n \times n$ MaxValue table and an $n \times n$ MinValue table, where $\mathsf{MaxValue}(i, j)$ corresponds to the cell at the i-th row and j-th column in the MaxValue table.

First, we establish the base cases. When $i = j$, since there is no operator and no parentheses to place, clearly $\mathsf{MaxValue}(i, i) = a_i$ and $\mathsf{MinValue}(i, i) = a_i$. (We can also initialize the cases when $i > j$ but this would be unnecessary since we will never reach these cases per our algorithm.)

We now present the following recursive relationship (or Bellman equation) for iteratively computing MaxValue and MinValue. For $j > i$,

$$\mathsf{MaxValue}(i,j) = \max_{i \leqslant k < j} \begin{cases} \mathsf{MaxValue}(i,k) + \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) - \mathsf{MinValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MinValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MinValue}(k+1,j) \end{cases} \tag{1}$$

$$\mathsf{MinValue}(i,j) = \min_{i \leqslant k < j} \begin{cases} \mathsf{MinValue}(i,k) + \mathsf{MinValue}(k+1,j) \\ \mathsf{MinValue}(i,k) - \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MaxValue}(i,k) \times \mathsf{MinValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MaxValue}(k+1,j) \\ \mathsf{MinValue}(i,k) \times \mathsf{MinValue}(k+1,j) \end{cases} \tag{2}$$

Thus, each recursive relationship requires considering 6 cases for each k. Computing the values for these tables should be done in a diagonal fashion, where we iterate from $|j - i| = 1$ to $n - 1$.

After computing the table of MaxValue and MinValue, the algorithm should return the answer at $\mathsf{MaxValue}(1, n)$.

(b)

We shall approach the proof of correctness using induction in a similar spirit of using induction in divide and conquer. Note that our recursive relationship uses smaller subproblems to compute the max and min values where "smaller" here refers to the smaller size intervals. Thus, we will induct on the size of the interval $j - i$ to prove that $\mathsf{MaxValue}(i, j)$ is the maximum value possible for the subsequence $a_i, \ldots, a_j$. We will also simultaneously prove that $\mathsf{MinValue}(i, j)$ is the minimum, as it is necessary to couple MaxValue and MinValue in the same induction because they are recursively dependent on each other.

1. **Base case**: $j - i = 0$

   When $i = j$, the maximum should simply be a_i as no operations are involved. The base cases we defined in our algorithm, $\mathsf{MaxValue}(i, i) = a_i$, reflect this observation, so $\mathsf{MaxValue}(i, i)$ is the maximum value possible. The same logic applies for $\mathsf{MinValue}(i, i)$.

2. **Inductive Hypothesis**: $j - i \leq k$

   Assume that all sub-problems $\mathsf{MaxValue}(i, j)$, where $j - i \leq k$, are indeed the maximum value possible for the subsequence $a_i, \ldots, a_j$. Assume the analagous for $\mathsf{MinValue}(i, j)$.

3. **Inductive Step**: $j - i = k + 1$

   The intuition to find $\mathsf{MaxValue}(i, j)$ or the maximum value possible achieved with numbers a_i through a_j is that we can try inserting parentheses at all possible positions and experimenting with $+, -,$and $\times$ to concatenate parenthesized sub group of numbers. In other words, with memoization, we are trying to concatenate

$$\begin{cases} \mathsf{MaxValue}(i,i) \text{ and } \mathsf{MaxValue}(i+1,j) \text{ with } +, -, \text{ and } \times \\ \mathsf{MaxValue}(i,i+1) \text{ and } \mathsf{MaxValue}(i+2,j) \text{ with } +, -, \text{ and } \times \\ \ldots \\ \mathsf{MaxValue}(i,j-1) \text{ and } \mathsf{MaxValue}(j,j) \text{ with } +, -, \text{ and } \times \end{cases}$$

   However, since the multiplication of two negative numbers can yield a positive number, we should also consider the case where we are concatenating the minimum value of two sub-expressions with $+, -,$and $\times$, and also the case where we are concatenating the min value of one sub-expression and the max value the other. The main reason why we only consider the operations between maximum and minimum values (and no value in between) is because of the following fact.

   Claim 1. *Consider $a_1 \in [l_1, r_1]$ and $a_2 \in [l_2, r_2]$ and applying a binary operation in that order. The maximum sum possible is $r_1 + r_2$ and the maximum difference is $r_1 - l_2$. The maximum product is* $\max\{l_1 l_2, l_1 r_2, r_1 l_2, r_1 r_2\}$.

   Proof. It is clear that the maximum sum should be adding the maximum of a_1 and a_2. Similarly, the maximum difference is straightforward. The maximum product cannot be anything other than the product of the extreme values of the intervals. This is because, if the maximum product were $a_1 a_2$ where $a_1 \neq l_1, r_1$, then you can always increase or decrease a_1 so that the product increases. □

   Due to the claim above, for a fixed middle-index k, we only need to consider 6 cases in total (1 for $+$, 1 for $-$, and 4 for $\times$), so we do not have to perform all the concatenations we did above, such as $\mathsf{MaxValue}(i, k) + \mathsf{MinValue}(k, j)$ and etc.

   Thus, $\mathsf{MaxValue}(i, j)$, where $j - i = k + 1$, is correctly computed by considering all the cases as in part (a), since all sub-problem calculations of interval size $\leq k$ are correct by the inductive hypothesis. The proof for the correctness of $\mathsf{MinValue}(i, j)$ follows similarly.

We can analyze the time complexity for the dynamic program as follows. The runtime is equal to the number of unique sub-problems $\times$ the time complexity per sub-problem, which is $O(n^2 \cdot n)$ = $O(n^3)$.

Problem 2

(a)

Given that part (b) asks for an algorithm that deals with a more general, we may naively apply the $O(n \log n + nH)$-runtime algorithm for part (a). However, this algorithm is not efficient because it has a pseudo-polynomial runtime. If we are given the information that all box heights $h_i = 1$, then we can use this to our advantage by devising an efficient $O(n \log n)$ algorithm based on a greedy strategy.

Side note: Actually this problem is remarkably similar to Problem 1 from Homework 4! Recall that in this problem we had to find the optimal order of deliveries for items i with profit p_i and due date d_i. In fact, we reduce our attic problem with $h_i = 1$ to this delivery problem by setting the profits $p_i = v_i$ and the due date $d_i = \lfloor H - w_i \rfloor$. Since we know a $O(n^2)$ algorithm to solving the delivery problem, in turn, we can solve part (a) in $O(n^2)$ time as well.

Algorithm 1: Arranging boxes with height of 1

```
Input: attic height $H$ and $n$ boxes with box $i$ having width $w_i$, height 1 and value $v_i$.
1 Sort the boxes from highest to lowest value and denote this list of boxes as $L$.
2 Let $G$ be the list of boxes added to shelf so far, keeping it sorted by decreasing width.
3 for box $i \in L$ do
4     Let $G'$ be $G$ with box $i$ added. Use binary search to keep $G'$ sorted by width.
5     if $G'$ is feasible then
6         Let $G \leftarrow G'$
7 return $G$
```

Although this is a valid idea, for the sake of completeness, we elaborate on the details of the algorithm without using reduction in the solution below.

To check feasibility, which means that there exists an order in the list of boxes G' such that those boxes fit in the attic, we can simply check if $w_{S[j]} \le H - j$ where j denotes the j-th element of G'. This is because it is always best to stack boxes with the widest widths first. Consider two consecutive boxes in a feasible order. If it's the case that the top box is wider than the bottom box, it would only help us to switch the two placements since the wider box will have more room by moving below and the narrower box can fit above.

Runtime For the runtime, checking feasibility for a list sorted by width takes linear time, while adding a box into G takes $O(\log n)$ time. Since the outer for loop has at most n iterations, the overall time complexity is $O(n^2)$.

Proof of correctness Denote the greedy solution above as G, and some valid and optimal solution as S. The goal is to prove that G is a valid and optimal list.

Firstly, order boxes by their value in decreasing order in S and G respectively, and relabel them as $S_1 ... S_n$ and $G_1 ... G_n$. If S and G choose the same boxes, G is optimal. If S is not the same as G, we can find the first box that S and G disagree on (S_i differs from G_i). If $v_{G_i} < v_{S_i}$, this wouldn't happen because boxes that fit in attic and have higher values would otherwise be chosen in our greedy algorithm output G. So since $v_{G_i} \ge v_{S_i}$, we would like to replace a box with an equal or lower value in S with G_i.

First, we add box G_i to S. If S remains feasible after the addition, then we found a solution that has higher value than the original S. This cannot be true so the new S must be infeasible, i.e. cannot fit in the attic. Consider the variable $l_i = \lfloor H - w_i \rfloor$ for box i. l_i denotes the highest level box i can be placed. In other words, i cannot be placed on top of l_i boxes. A set of boxes is feasible if and only if for all levels l that $l \le n$, the number of boxes that need to placed under or at level l is no more than l (similarly to HW4 Problem 1). Since adding box G_i made S infeasible, there exists a level l where the number of boxes that need to be placed under or at level l is equal to $l + 1$. There must exist a box with value less than or equal to v_{G_i} here since if it were only boxes from $\{1 \dots i - 1\}$, S would be feasible. Thus, we can exchange this box with G_i and create a set of boxes of greater or equal value than S. If we keep resolving the differences between S and G following the above procedure, S will finally be transformed into G as there's only a finite number of boxes in total.

(b)

First, sort the list of boxes so that the boxes are decreasing in width, i.e. $w_1 \ge w_2 \ge \cdots \ge w_n$. We define $\mathsf{OPT}(i, h)$ as the maximum value we could obtain using a subset of items $\{1, 2, \dots, i\}$ and with a remaining height of the attic h. We can calculate it with the following relationship:

$$\mathsf{OPT}(i, h) = \begin{cases} \mathsf{OPT}(i-1, h) & \text{if } w_i > h - h_i \\ \max(\mathsf{OPT}(i-1, h), v_i + \mathsf{OPT}(i-1, h-h_i)) & \text{else} \end{cases} \tag{3}$$

For the base case, let $\mathsf{OPT}(0, h) = 0$. Given the recursive relation and the base cases, our algorithm iteratively computes OPT and finally outputs the answer $\mathsf{OPT}(n, H)$.

Algorithm 2: Arranging boxes with arbitrary integer heights

```
Input: attic height $H$ and $n$ boxes with box $i$ having width $w_i$, height $h_i$ and value $v_i$.
1 Sort the boxes by increasing order of width.
2 for $i = 1, 2, \ldots, n$ do
3     for $h = 1, 2, \ldots, H$ do
4         if $w_i \le h - h_i$ then
5             $\mathsf{OPT}(i, h) = \max(\mathsf{OPT}(i-1, h), v_i + \mathsf{OPT}(i-1, h-h_i))$
6         else
7             $\mathsf{OPT}(i, h) = \mathsf{OPT}(i-1, h)$
8 return $\mathsf{OPT}(n, H)$.
```

(c)

Proof of correctness We can prove correctness via induction on i from 0 to n.

1. **Base case**: $i = 0$

   $i = 0$ means there's no boxes to add to attic so the maximum value $\mathsf{OPT}(i, h) = 0$ for any $h \in [0, H]$.

2. **Inductive hypothesis**: $i \le k$

   $\mathsf{OPT}(i, h)$ represents the maximum value achievable for boxes $\{1, \ldots, i\}$ (sorted by increasing width) and attic height h where $i \le k$ and $0 \le h \le H$.

3. **Inductive step**: $i = k + 1$

   In order to find $\mathsf{OPT}(k+1, h)$, or the maximum value obtained using a subset of box$_1$ through box$_{k+1}$ and with a remaining height of the attic h, we can consider breaking this problem down exhaustively into two sub-problems depending on whether we add box$_{k+1}$ to the attic or not.

   Case 1 If we add box$_{k+1}$ to attic, we would gain v_{k+1} in value. Since w_{k+1} is at least as large as $w_1 ... w_k$, we know that putting w_{k+1} at the bottom of an attic of height h would yield the most efficient allocation of space, as swapping box$_{k+1}$ with any other selected box from box$_1$...box$_k$ might lead to an infeasible set and thus a lower value. Then the remaining height for selecting from box$_1$ through box$_k$ would be $h - h_{k+1}$, and the sub-problem we are solving becomes v_{k+1} + $\mathsf{OPT}(k, h - h_i)$, where $\mathsf{OPT}(k, h - h_i)$ is correct by the inductive hypothesis.

   Case 2 If we don't add box$_{k+1}$ to attic, we would not be able to gain v_{k+1} in value and the remaining height for selecting from box$_1$ through box$_k$ would still be h. Then the sub-problem we are solving becomes $\mathsf{OPT}(k, h)$, where $\mathsf{OPT}(k, h)$ is correct by the inductive hypothesis.

   Thus, we know that each case is computed as desired, and thus taking the max of the two cases gives $\mathsf{OPT}(k + 1, h)$. Of course, if box$_{k+1}$ has a h_{k+1} larger than h, we would be forced to give up box$_{k+1}$ and ends up in case 2 above.

Time Complexity Sorting takes $O(n \log n)$ and the two for loops take $O(nH)$. Thus the runtime is $O(n \log n + nH)$.