Graded Exercise 3

Duong Le

Problem 1

a.

Case 1. $k \notin S_k^*$

Suppose that $V(k-1, w)$ is not the optimum solution for the problem (k, w). This means that there exists an optimal solution $V'(k, w)$ such that $V'(k, w) > V(k-1, w)$. However, since $k \notin S_k^*$, we are not adding any excessive value to the knapsack when adding or removing k from the problem.

$$\Rightarrow V'(k, w) = V'(k-1, w) > V(k, w)$$

We arrive at a contradiction since we know that $V(k, w)$ is the optimum solution for (k, w).

Case 2. $k \in S_k^*$

Suppose agian that $V(k-1, w-w_k) + v_k$ is not the optimum solution for the problem (k, w). This means that there exists an optimal solution $V'(k, w)$ such that $V'(k, w) > V(k-1, w) + v_k$. Since $k \in S_k^*$, we can remove k from the problem, it follows that.

$$V'(k-1, w-w_k) - v_k > V(k-1, w-w_k)$$

We also arrive at a contradiction since we know that $V(k-1, w-w_k)$ is the optimum solution for $(k-1, w-w_k)$.

$\Longrightarrow$ From the two cases, we can verified that (1) is correct.

b.

Case 1. $w_k > w$

Since $w_k > w$, the item k cannot be added to the knapsack, or we can say that $k \notin S_k^*$. In the previous part, we have proved that $S_k^* = V(k-1, w)$.
$\Rightarrow V(k, w) = V(k-1, w)$ if $w_k > w$

Case 2. $w_k \leq w$

In this case, there are two possibilities: either $k \notin S_k^*$ or $k \in S_k^*$. It also follow from the previous part that solutions for the two possibilities are $V(k, w)$ and $V(k-1, w-w_k) + v_k$ respectively. And since the problem is to find the maximum value.
$\Rightarrow V(k, w) = max\{V(k-1, w), V(k-1, w-w_k) + v_k\}$ if $w_k \leq w$

$\implies$ From the two cases, we can verified that (2) is correct.

c.

Algorithm.

From the two part, we can fomulate the following algorithms, with the method $fill\_table$ used to fill an array of size nC with the knapsack values, and the method *knapsack* to compute and return the maximum value of the knapsack problem:

Algorithm 1 $fill\_table(n, C, arr)$

```
for $w \leftarrow 0$ to $C$ do
    for $k \leftarrow 0$ to $n$ do
        if $(k = 0)$ then
            $arr[k][w] \leftarrow 0$
        else if $(w_k > w)$ then
            $arr[k][w] \leftarrow arr[k-1][w]$
        else
            $arr[k][w] \leftarrow \max\{arr[k-1][w], arr[k-1][w-w_k] + value(k)\}$
        end if
    end for
end for
```

Algorithm 2 $knapsack(n, C)$

```
$arr \leftarrow array[n+1][C+1]$        ▷ Create a two dimensional array to store the value.
$fill\_table(n, C, arr)$
return $arr[n][C]$
```

Memory and time complexity.

Consider, the function *knapsack* only allocate one array of size nC and then store the values inside. Furthermore, the function $fill\_table$ does not use any additional amount of space.
$\Rightarrow$ The function uses $O(nC)$ memory.

Next, consider the function $fill\_table$, when call using n items and weight capacity C, the function will loop nC times. In each loop, the function will perform at most 3 comparisions, 1 addition, and 2 subtraction. The 3 comparision is achieved in the last *else*, when the function compares k to zero, the weigth of k with w, and comparing the results of two sub-problems.

$$\rightarrow T(n, C) = nC(3O(1) + O(1) + 2O(1)) = O(nC)$$

Now, if we assume that allocate an array of size nC takes $O(nC)$ times and accessing an element takes $O(1)$, we get the time complexity of *knapsack*:

$$T(n, C) = O(nC) + O(nC) + O(1) = O(nC)$$

$\Rightarrow$ The function runs in $O(nC)$ time.

$\Longrightarrow$ The function *knapsack* runs in $O(nC)$ time and memory.

d.

Consider:

$$\begin{aligned} w_i &\leq U \\ \Leftrightarrow w - w_i &\geq w - U \end{aligned}$$

So when updating the array during a weight iteration, we never use more than U previous weight cells. So the modification methods will be the following. We use the modulo operator to get the position in the array since the modulo can iterate all the cells in the array of size U.

Algorithm 3 $fill\_table(n, C, U, arr)$

```
for $w \leftarrow 0$ to $C$ do
    $w_1 \leftarrow w\%U$
    for $k \leftarrow 0$ to $n$ do
        if $(k = 0)$ then
            $arr[k][w1] \leftarrow 0$
        else if $(w_k > w)$ then
            $arr[k][w_1] \leftarrow arr[k-1][w_1]$
        else
            $w_2 \leftarrow (w - w_k)\%U$
            $arr[k][w_1] \leftarrow \max\{arr[k-1][w_1], arr[k-1][w_2] + value(k)\}$
        end if
    end for
end for
```

Algorithm 4 $knapsack(n, C, U)$

```
$arr \leftarrow array[n+1][U+1]$        ▷ Create a two dimensional array to store the value.
$fill\_table(n, C, U, arr)$
return $arr[n][U]$
```

It is clear that the algorithm runs in $O(nU)$ memory (since we use an array of size nU to store the intergers), and the correctness has been proven above. So the presented algorithm satisfies the requirements.